MC-202
Backtracking

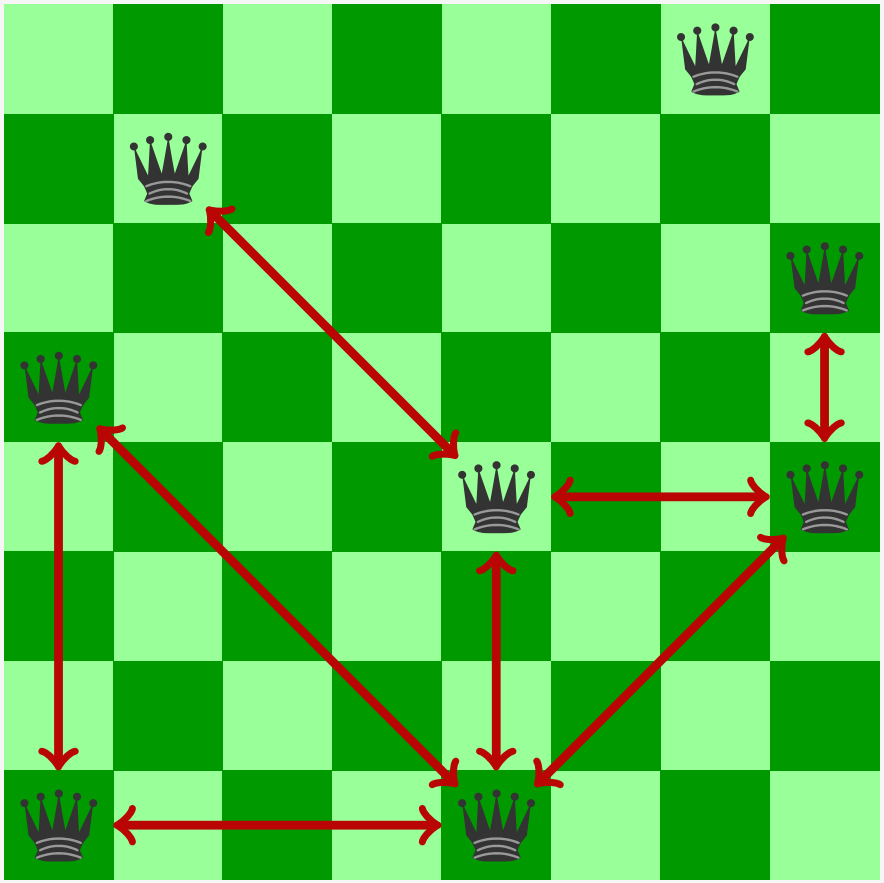
Lehilton Pedrosa

Universidade Estadual de Campinas

Segundo semestre de 2022

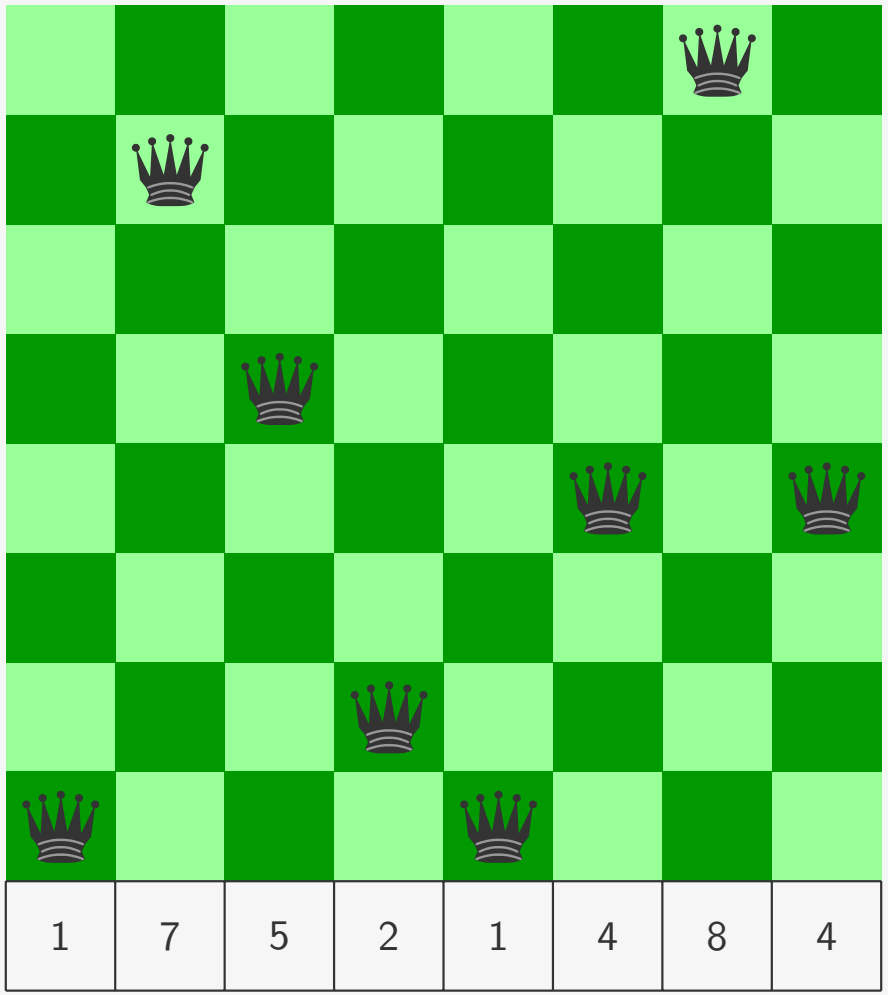
Um problema

Como dispor oito damas em um tabuleiro de xadrez, sem posições de ameaça?



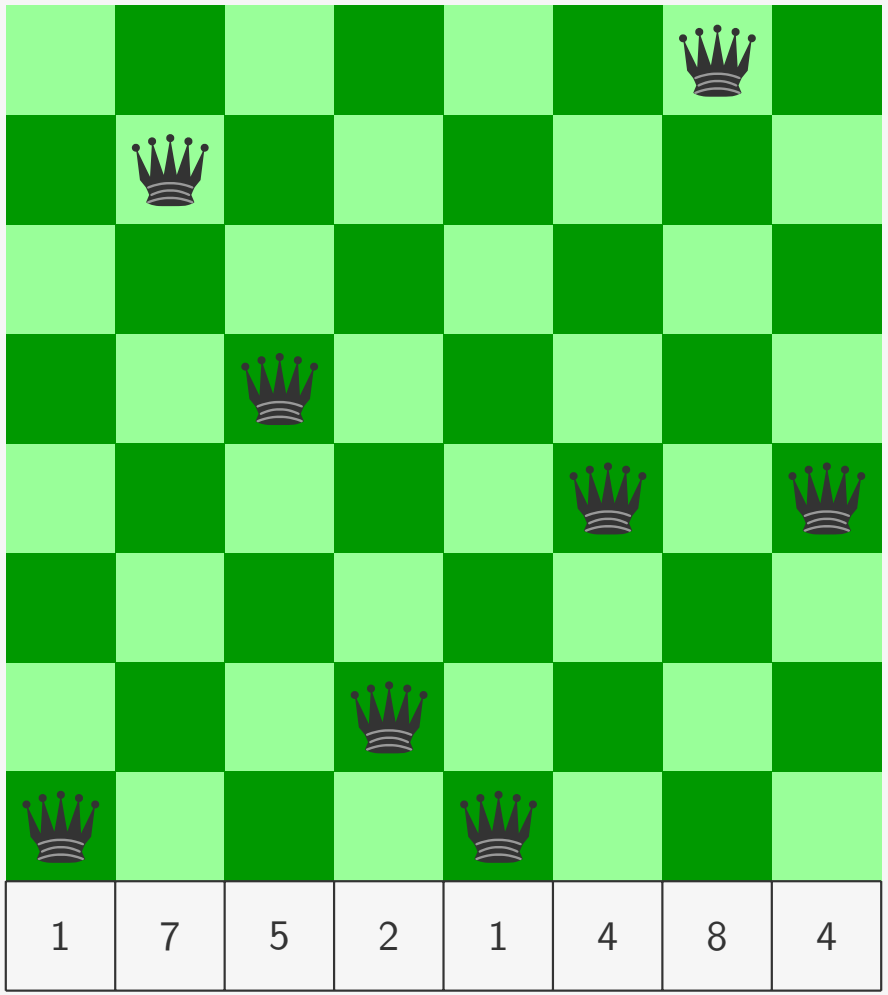
Testando (quase) todas as soluções

- Cada coluna dever ter exatamente uma dama

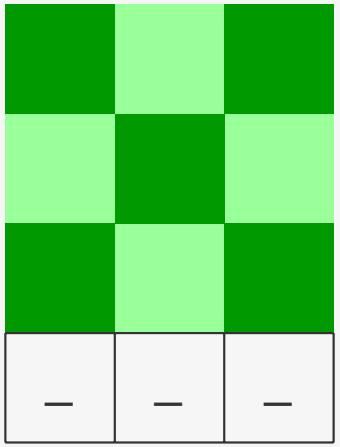


Testando (quase) todas as soluções

- Cada coluna dever ter exatamente uma dama
- Representamos uma disposição com um vetor

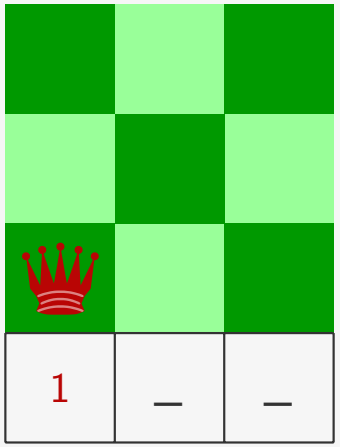


Enumerando



Enumerando disposições de tamanho 3:

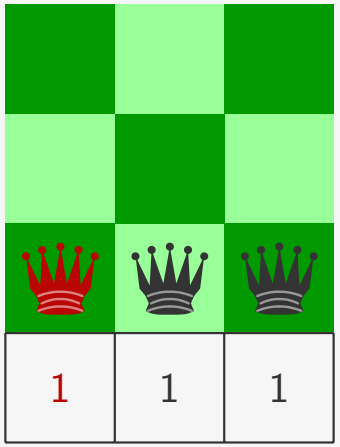
Enumerando



Enumerando disposições de tamanho 3:

1. fixamos a primeira posição

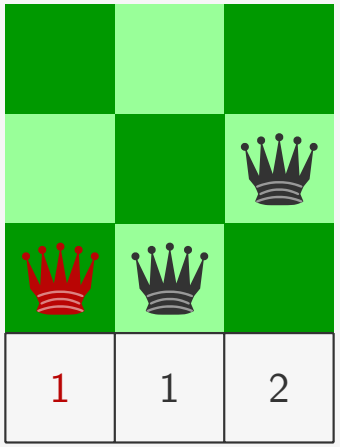
Enumerando



Enumerando disposições de tamanho 3:

1. fixamos a primeira posição
2. listamos todas os sufixos de tamanho 2

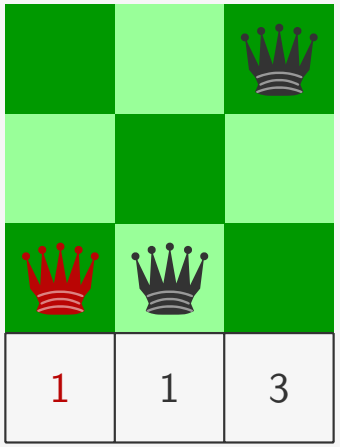
Enumerando



Enumerando disposições de tamanho 3:

1. fixamos a primeira posição
2. listamos todas os sufixos de tamanho 2

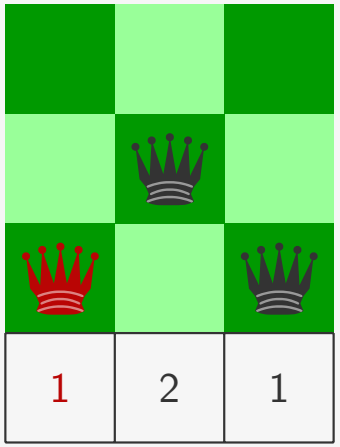
Enumerando



Enumerando disposições de tamanho 3:

1. fixamos a primeira posição
2. listamos todas os sufixos de tamanho 2

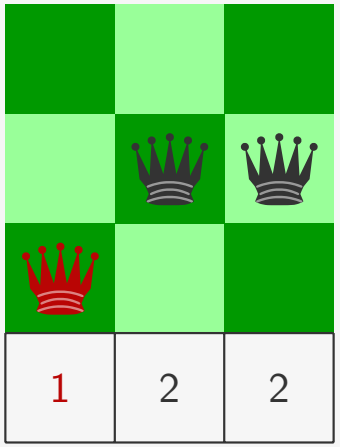
Enumerando



Enumerando disposições de tamanho 3:

1. fixamos a primeira posição
2. listamos todas os sufixos de tamanho 2

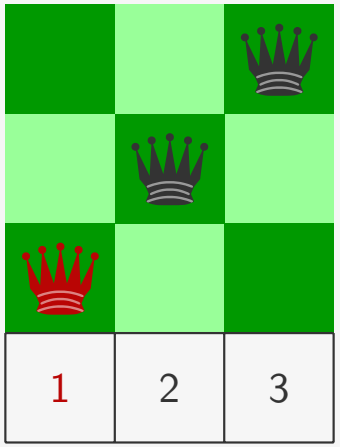
Enumerando



Enumerando disposições de tamanho 3:

1. fixamos a primeira posição
2. listamos todas os sufixos de tamanho 2

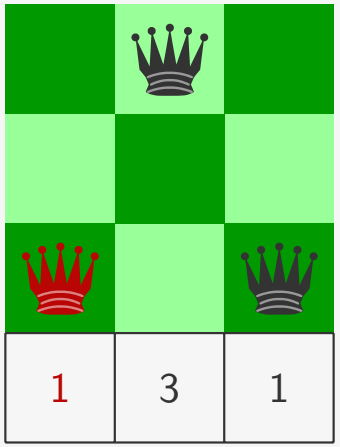
Enumerando



Enumerando disposições de tamanho 3:

1. fixamos a primeira posição
2. listamos todas os sufixos de tamanho 2

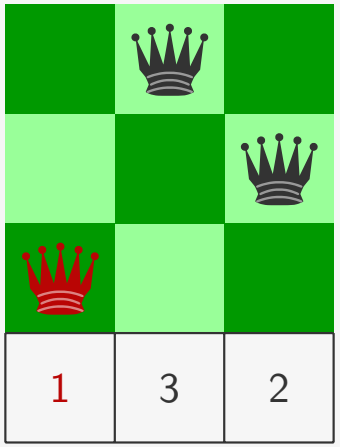
Enumerando



Enumerando disposições de tamanho 3:

1. fixamos a primeira posição
2. listamos todas os sufixos de tamanho 2

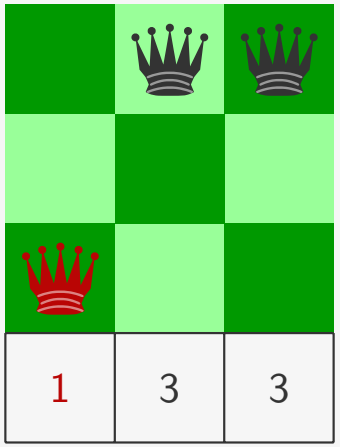
Enumerando



Enumerando disposições de tamanho 3:

1. fixamos a primeira posição
2. listamos todas os sufixos de tamanho 2

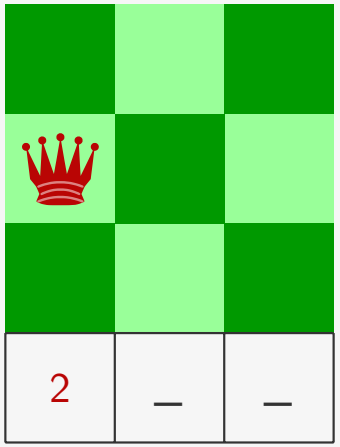
Enumerando



Enumerando disposições de tamanho 3:

1. fixamos a primeira posição
2. listamos todas os sufixos de tamanho 2

Enumerando



Enumerando disposições de tamanho 3:

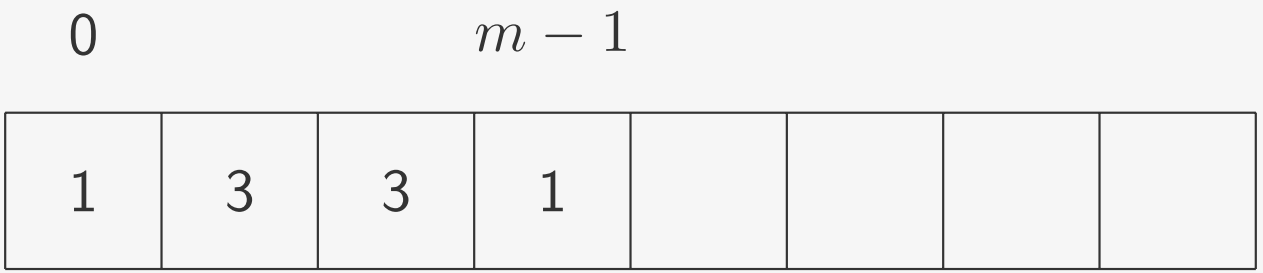
1. fixamos a primeira posição
2. listamos todas os sufixos de tamanho 2
3. repetimos para as outras possibilidades

Enumerando em geral

Como imprimir todas as disposições com prefixo dado?

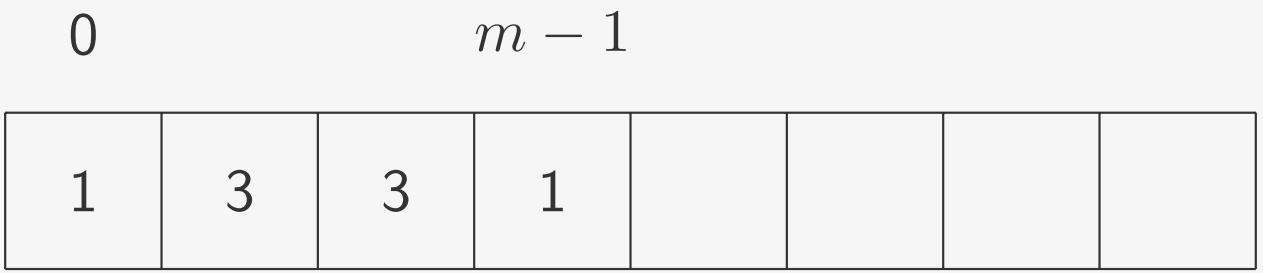
Enumerando em geral

Como imprimir todas as disposições com prefixo dado?



Enumerando em geral

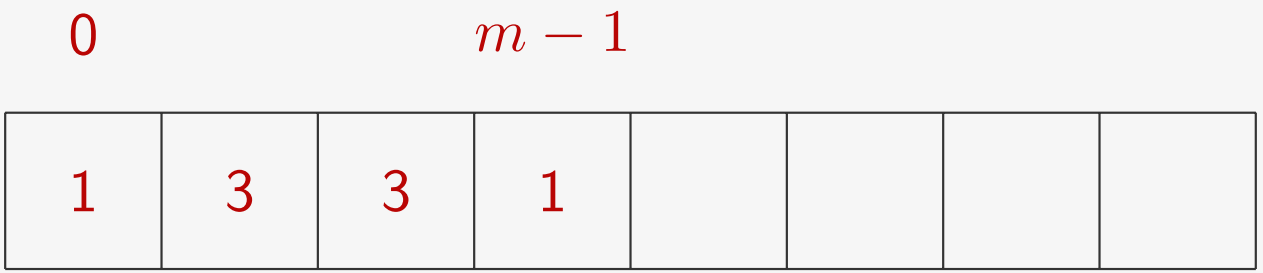
Como imprimir todas as disposições com prefixo dado?



Vamos escrever uma função que receba um vetor:

Enumerando em geral

Como imprimir todas as disposições com prefixo dado?

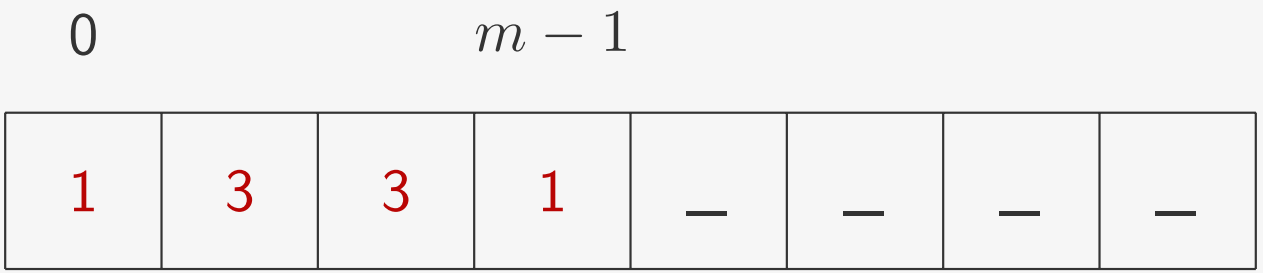


Vamos escrever uma função que receba um vetor:

1. com valores fixos até uma posição $m-1$

Enumerando em geral

Como imprimir todas as disposições com prefixo dado?



Vamos escrever uma função que receba um vetor:

1. com valores fixos até uma posição $m-1$
2. com posições abertas de m até $n-1$

Programamando

Como imprimir todas disposições (recursivamente)

```
1 void enumerar(int vetor[], int m, int n) {
```

Programando

Como imprimir todas disposições (recursivamente)

```
1 void enumerar(int vetor[], int m, int n) {
2   // se todas posições estão fixas, só há uma combinação
3   if (n == m) {
4     imprimir_vetor(vetor, n);
5     return ;
6   }
7
```

Programando

Como imprimir todas disposições (recursivamente)

```
1  void enumerar(int vetor[], int m, int n) {
2    // se todas posições estão fixas, só há uma combinação
3    if (n == m) {
4      imprimir_vetor(vetor, n);
5      return ;
6    }
7
8    // senão, estendemos o prefixo em uma posição
9    for (int i = 1; i <= n; i++) {
10     vetor[m] = i
11     enumerar(vetor, m + 1, n);
12   }
13 }
```

Encontrando uma solução

Como verificar se existe uma disposição válida das damas?

Encontrando uma solução

Como verificar se existe uma disposição válida das damas?

- `disposicao_valida` recebe um vetor preenchido

Encontrando uma solução

Como verificar se existe uma disposição válida das damas?

- `disposicao_valida` recebe um vetor preenchido
  - devolve `1` se nenhuma dama ataca outra

Encontrando uma solução

Como verificar se existe uma disposição válida das damas?

- `disposicao_valida` recebe um vetor preenchido
  - devolve `1` se nenhuma dama ataca outra
  - devolve `0` caso contrário

Encontrando uma solução

Como verificar se existe uma disposição válida das damas?

- `disposicao_valida` recebe um vetor preenchido
  - devolve `1` se nenhuma dama ataca outra
  - devolve `0` caso contrário

- `existe_solucao` recebe um vetor parcialmente preenchido com m posições fixas

Encontrando uma solução

Como verificar se existe uma disposição válida das damas?

- `disposicao_valida` recebe um vetor preenchido
  - devolve `1` se nenhuma dama ataca outra
  - devolve `0` caso contrário

- `existe_solucao` recebe um vetor parcialmente preenchido com m posições fixas
  - devolve `1` se existe alguma disposição válida que mantém a posição das m primeiras damas

Encontrando uma solução

Como verificar se existe uma disposição válida das damas?

- `disposicao_valida` recebe um vetor preenchido
  - devolve `1` se nenhuma dama ataca outra
  - devolve `0` caso contrário

- `existe_solucao` recebe um vetor parcialmente preenchido com m posições fixas
  - devolve `1` se existe alguma disposição válida que mantém a posição das m primeiras damas
  - devolve `0` caso contrário

Encontrando uma solução

Como verificar se existe uma disposição válida das damas?

- `disposicao_valida` recebe um vetor preenchido
  - devolve `1` se nenhuma dama ataca outra
  - devolve `0` caso contrário

- `existe_solucao` recebe um vetor parcialmente preenchido com m posições fixas
  - devolve `1` se existe alguma disposição válida que mantém a posição das m primeiras damas
  - devolve `0` caso contrário

Exercício: implemente `disposicao_valida`

Enumerando e testando

Existe solução com as primeiras damas já dispostas?

```
1 int existe_solucao(int vetor[], int m, int n) {
```

Enumerando e testando

Existe solução com as primeiras damas já dispostas?

```
1  int existe_solucao(int vetor[], int m, int n) {
2    if (n == m) {
3      if (disposicao_valida(vetor, n)) {
4        imprimir_vetor(vetor, n);
5        return 1;
6      } else {
7        return 0;
8      }
9    }
10
```

Enumerando e testando

Existe solução com as primeiras damas já dispostas?

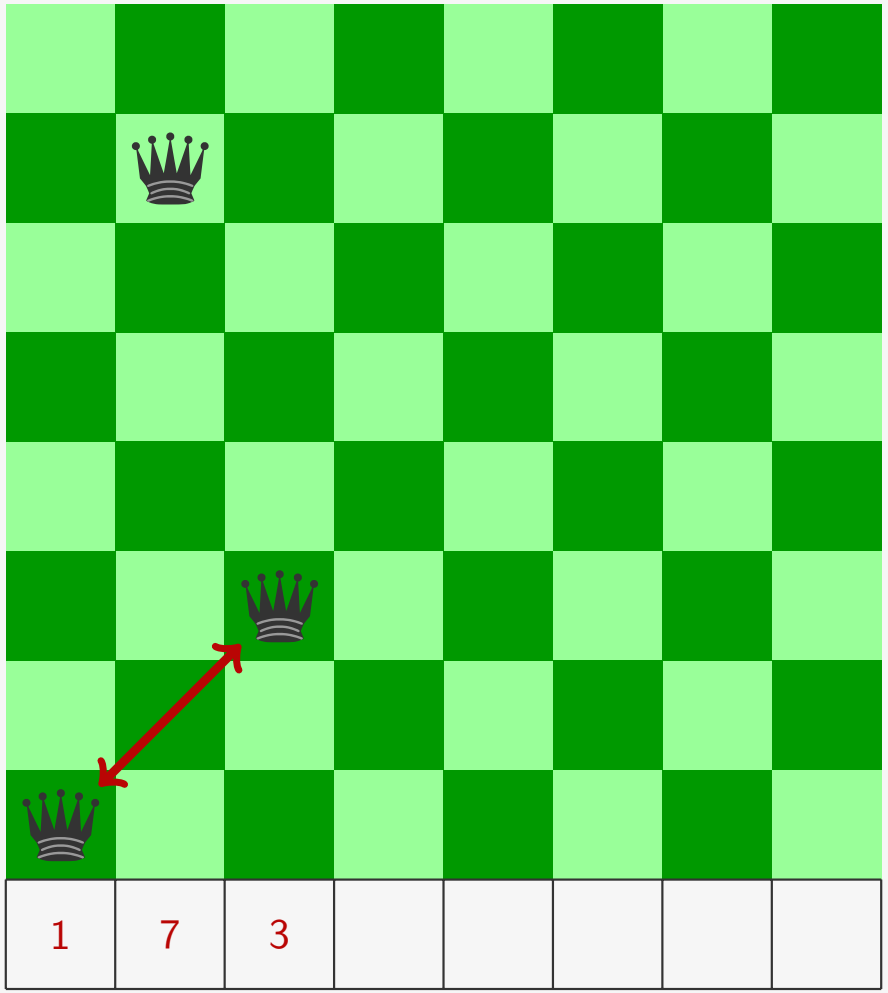
```
1  int existe_solucao(int vetor[], int m, int n) {
2    if (n == m) {
3      if (disposicao_valida(vetor, n)) {
4        imprimir_vetor(vetor, n);
5        return 1;
6      } else {
7        return 0;
8      }
9    }
10
11   for (int i = 1; i <= n; i++) {
12     vetor[m] = i;
13     if (existe_solucao(vetor, m + 1, n))
14       return 1;
15   }
16
```

Enumerando e testando

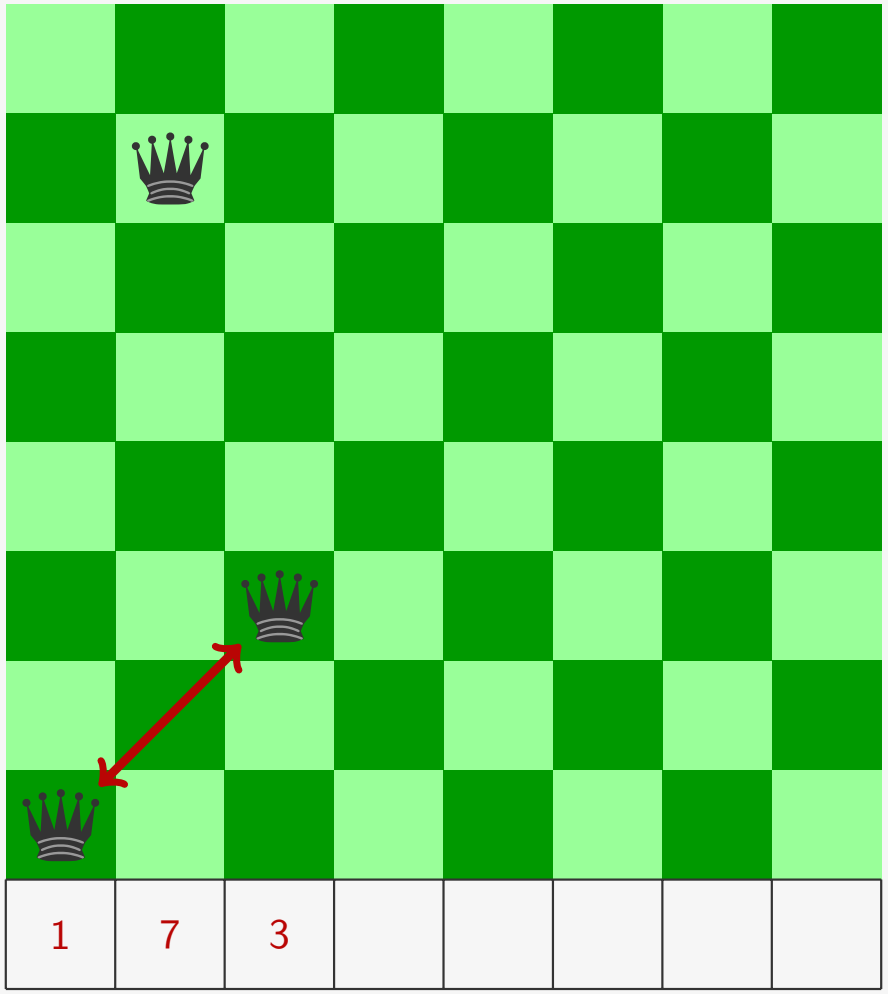
Existe solução com as primeiras damas já dispostas?

```
1  int existe_solucao(int vetor[], int m, int n) {
2    if (n == m) {
3      if (disposicao_valida(vetor, n)) {
4        imprimir_vetor(vetor, n);
5        return 1;
6      } else {
7        return 0;
8      }
9    }
10
11   for (int i = 1; i <= n; i++) {
12     vetor[m] = i;
13     if (existe_solucao(vetor, m + 1, n))
14       return 1;
15   }
16
17   return 0;
18 }
```

Melhorando um pouco

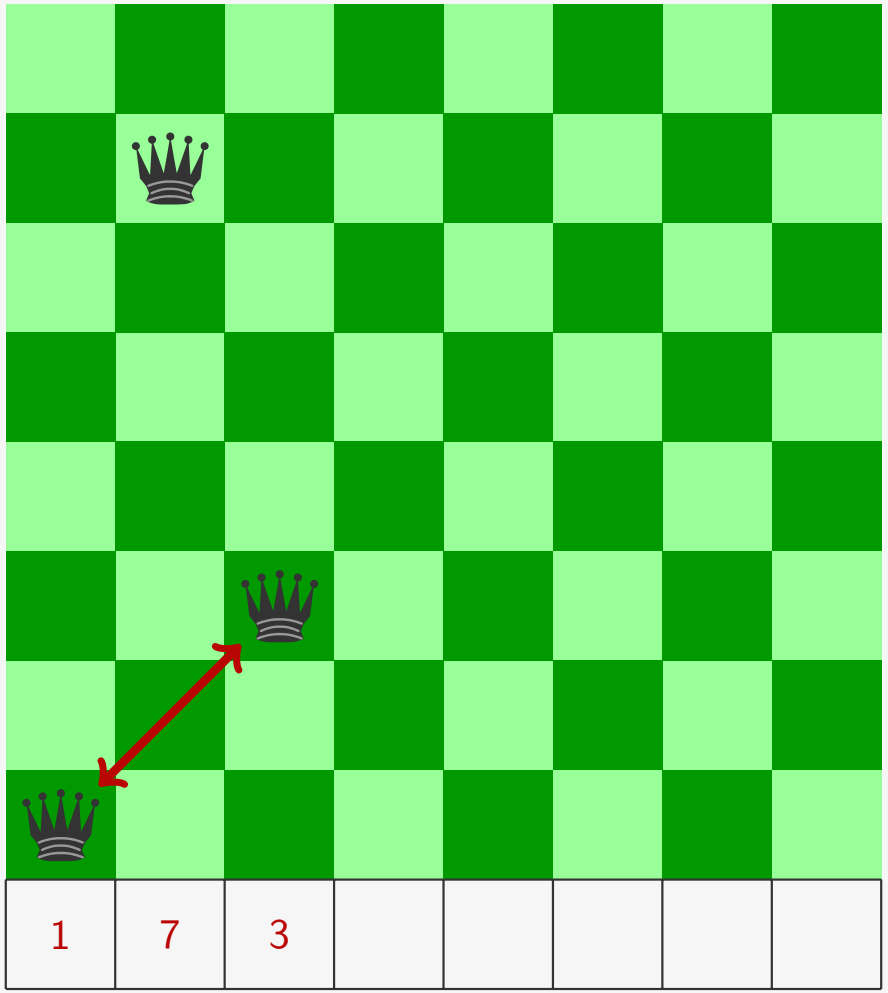


Melhorando um pouco



- alguns prefixos não são viáveis

Melhorando um pouco



- alguns prefixos não são viáveis
- não precisamos testar disposições com esses prefixos

Prefixo viável

Suponha que

Prefixo viável

Suponha que

- `prefixo_viavel` recebe um vetor parcialmente preenchido

Prefixo viável

Suponha que

- `prefixo_viavel` recebe um vetor parcialmente preenchido
  - devolve `1` se nenhuma dama do prefixo ataca outra

Prefixo viável

Suponha que

- `prefixo_viavel` recebe um vetor parcialmente preenchido
  - devolve `1` se nenhuma dama do prefixo ataca outra
  - devolve `0` caso contrário

Ignorando prefixos inviáveis

Se um prefixo é inviável, não continuamos a recursão!

```
1 int existe_solucao(int vetor[], int m, int n) {
```

Ignorando prefixos inviáveis

Se um prefixo é inviável, não continuamos a recursão!

```
1 int existe_solucao(int vetor[], int m, int n) {
2   if (!prefixo_viavel(vetor, m))
3       return 0;
4
```

Ignorando prefixos inviáveis

Se um prefixo é inviável, não continuamos a recursão!

```
1 int existe_solucao(int vetor[], int m, int n) {
2   if (!prefixo_viavel(vetor, m))
3       return 0;
4
5   if (n == m)
6       return 1;
7
```

Ignorando prefixos inviáveis

Se um prefixo é inviável, não continuamos a recursão!

```
1  int existe_solucao(int vetor[], int m, int n) {
2    if (!prefixo_viavel(vetor, m))
3        return 0;
4
5    if (n == m)
6        return 1;
7
8    for (int i = 1; i <= n; i++) {
9      vetor[m] = i;
10     if (existe_solucao(vetor, m + 1, n))
11       return 1;
12   }
13
```

Ignorando prefixos inviáveis

Se um prefixo é inviável, não continuamos a recursão!

```
1  int existe_solucao(int vetor[], int m, int n) {
2    if (!prefixo_viavel(vetor, m))
3        return 0;
4
5    if (n == m)
6        return 1;
7
8    for (int i = 1; i <= n; i++) {
9      vetor[m] = i;
10     if (existe_solucao(vetor, m + 1, n))
11       return 1;
12   }
13
14   return 0;
15 }
```

Testando prefixo

```
1 int prefixo_viavel(int vetor[], int m) {
```

Testando prefixo

```
1 int prefixo_viavel(int vetor[], int m) {
2   // para cada coluna do prefixo
3   for (int i = 0; i < m - 1; i++) {
```

Testando prefixo

```
1 int prefixo_viavel(int vetor[], int m) {
2   // para cada coluna do prefixo
3   for (int i = 0; i < m - 1; i++) {
4
5     // se está na mesma linha
6     if (vetor[i] == vetor[m-1])
7       return 0;
```

Testando prefixo

```
1  int prefixo_viavel(int vetor[], int m) {
2    // para cada coluna do prefixo
3    for (int i = 0; i < m - 1; i++) {
4
5      // se está na mesma linha
6      if (vetor[i] == vetor[m-1])
7        return 0;
8
9      // se está na mesma diagonal
10     if ((m - 1) - i == abs(vetor[m-1] - vetor[i]))
11       return 0;
```

Testando prefixo

```
1  int prefixo_viavel(int vetor[], int m) {
2    // para cada coluna do prefixo
3    for (int i = 0; i < m - 1; i++) {
4
5      // se está na mesma linha
6      if (vetor[i] == vetor[m-1])
7        return 0;
8
9      // se está na mesma diagonal
10     if ((m - 1) - i == abs(vetor[m-1] - vetor[i]))
11       return 0;
12   }
13
14   return 1;
15 }
```

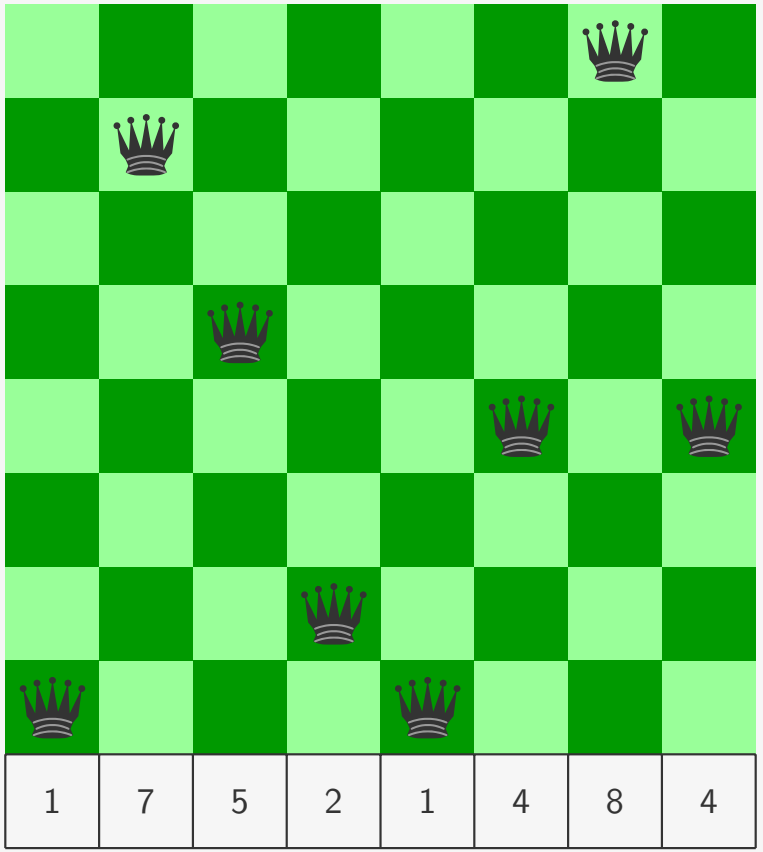
Testando prefixo

```
1  int prefixo_viavel(int vetor[], int m) {
2    // para cada coluna do prefixo
3    for (int i = 0; i < m - 1; i++) {
4
5      // se está na mesma linha
6      if (vetor[i] == vetor[m-1])
7        return 0;
8
9      // se está na mesma diagonal
10     if ((m - 1) - i == abs(vetor[m-1] - vetor[i]))
11       return 0;
12   }
13
14   return 1;
15 }
```

Pergunta: por que só precisamos comparar o último elemento do prefixo com os anteriores?

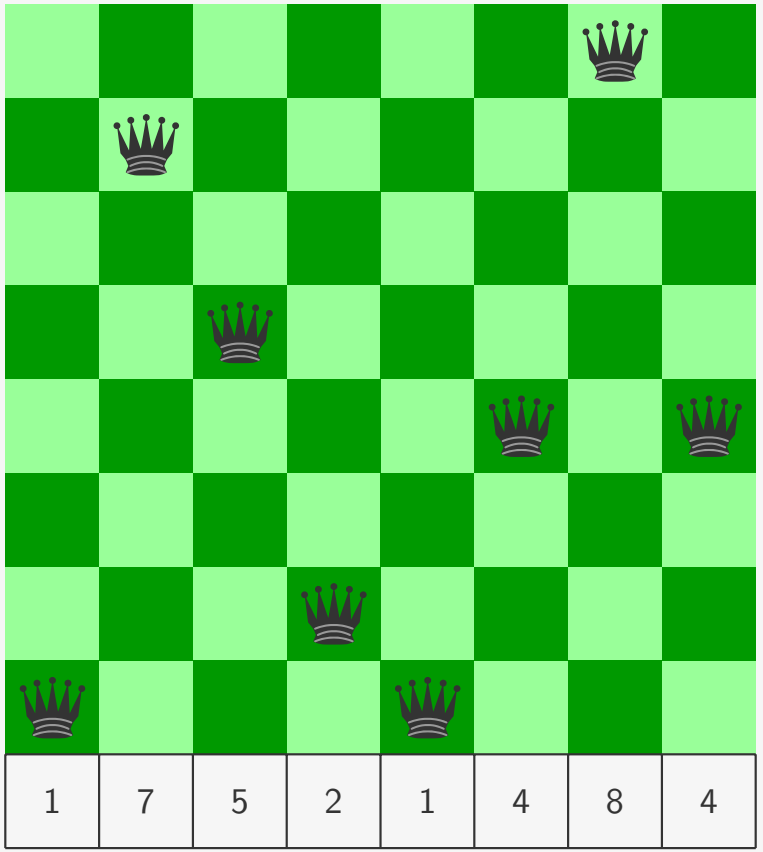
Modificando a estratégia

Como diminuir as disposições testadas?



Modificando a estratégia

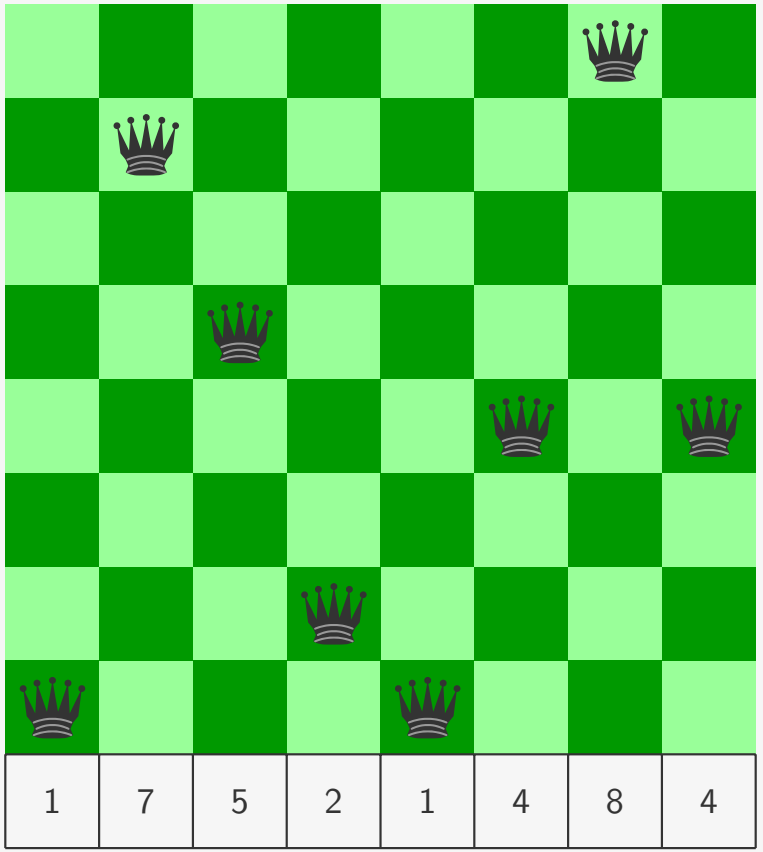
Como diminuir as disposições testadas?



- cada coluna só deve ter uma dama:

Modificando a estratégia

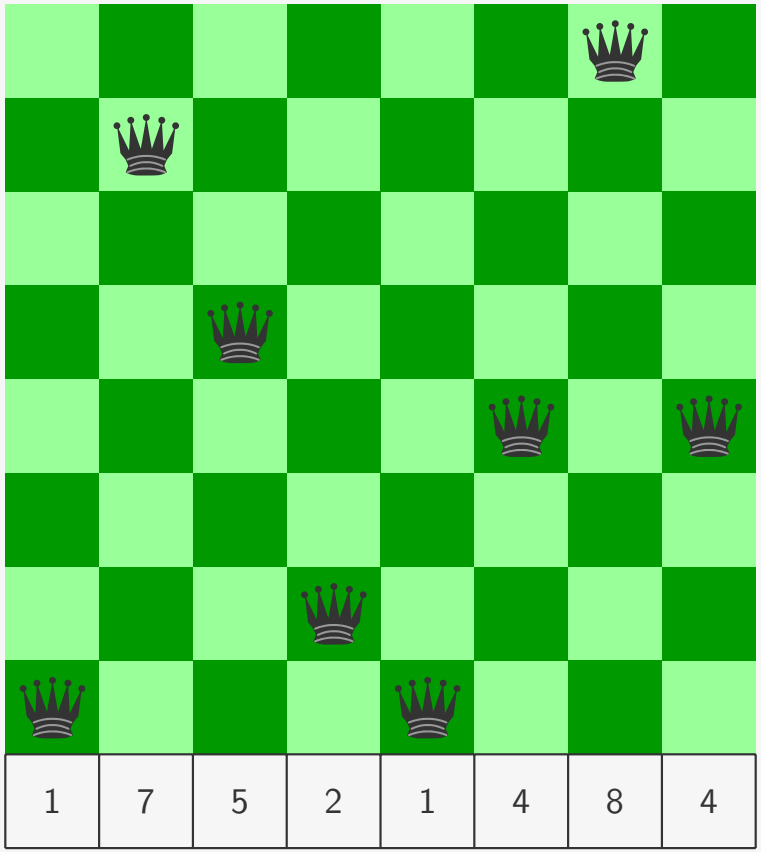
Como diminuir as disposições testadas?



- cada coluna só deve ter uma dama: ✓

Modificando a estratégia

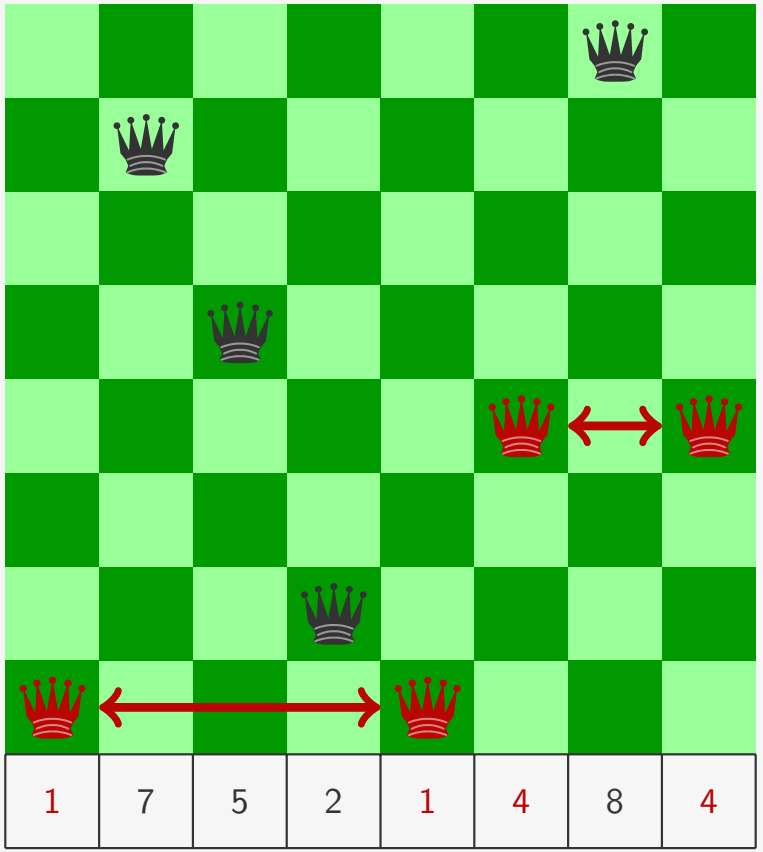
Como diminuir as disposições testadas?



- cada coluna só deve ter uma dama: ✓
- cada linha só deve ter uma dama:

Modificando a estratégia

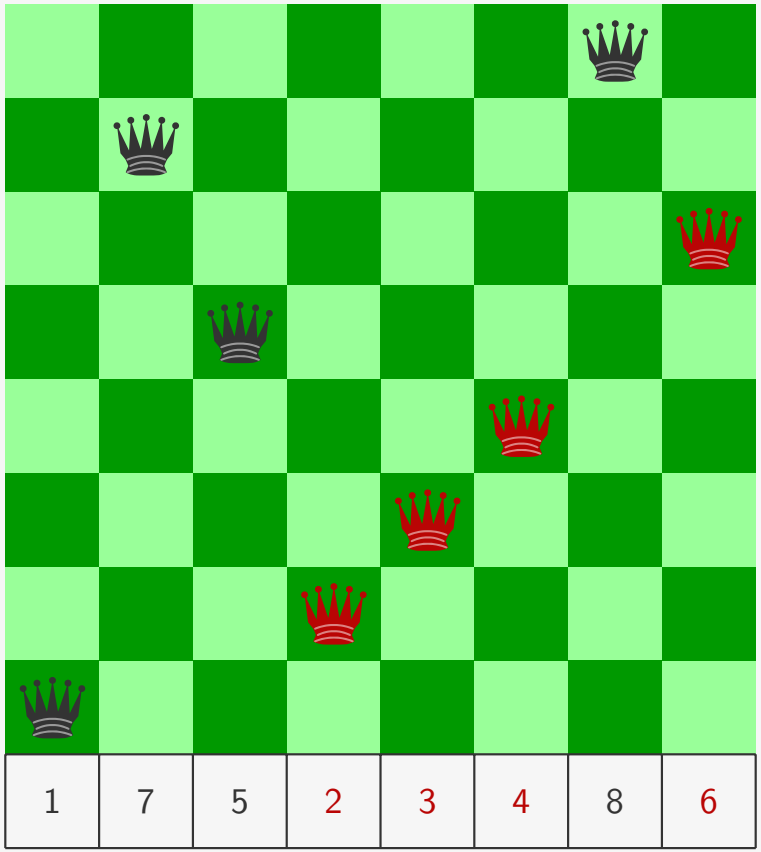
Como diminuir as disposições testadas?



- cada coluna só deve ter uma dama: ✓
- cada linha só deve ter uma dama: ✗

Modificando a estratégia

Como diminuir as disposições testadas?



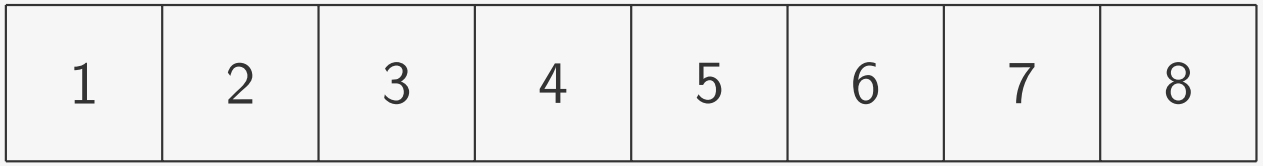
- cada coluna só deve ter uma dama: ✓
- cada linha só deve ter uma dama: ✗

Observação: uma disposição deve ser uma permutação

Permutações

Vamos escreve uma função `permutacoes` que:

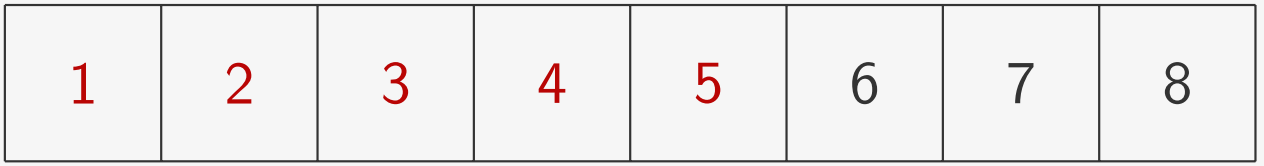
Permutações



Vamos escreve uma função `permutacoes` que:

- Recebe um vetor com n números distintos:

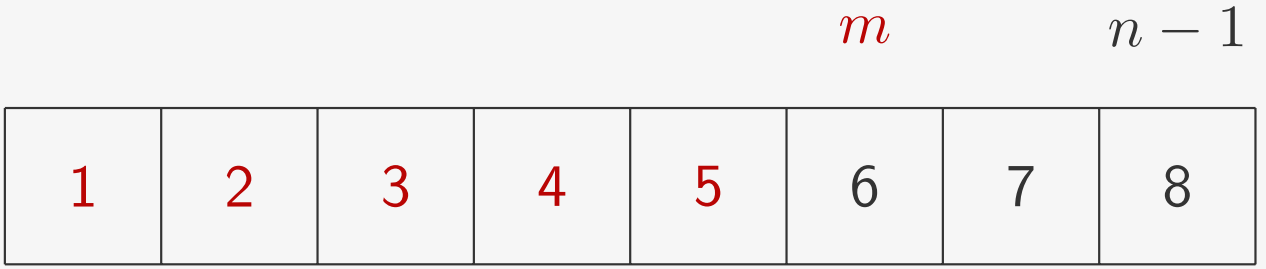
Permutações



Vamos escreve uma função `permutacoes` que:

- Recebe um vetor com n números distintos:
  - m primeiras posições devem ser mantidas

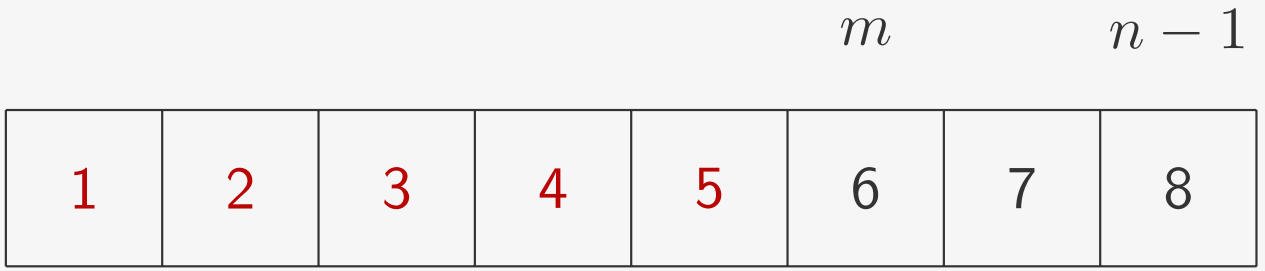
Permutações



Vamos escreve uma função `permutacoes` que:

- Recebe um vetor com n números distintos:
  - m primeiras posições devem ser mantidas
  - posições de m até $n-1$ devem ser permutadas

Permutações

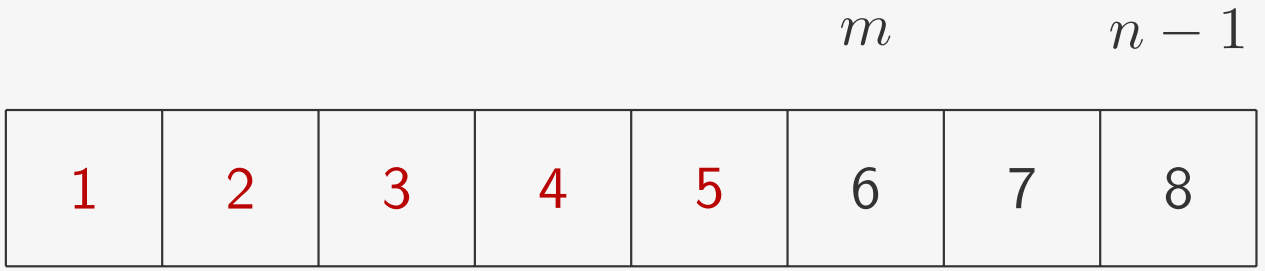


Vamos escreve uma função `permutacoes` que:

- Recebe um vetor com n números distintos:
  - m primeiras posições devem ser mantidas
  - posições de m até $n-1$ devem ser permutadas

- Imprime todas permutações que esse prefixo:

Permutações

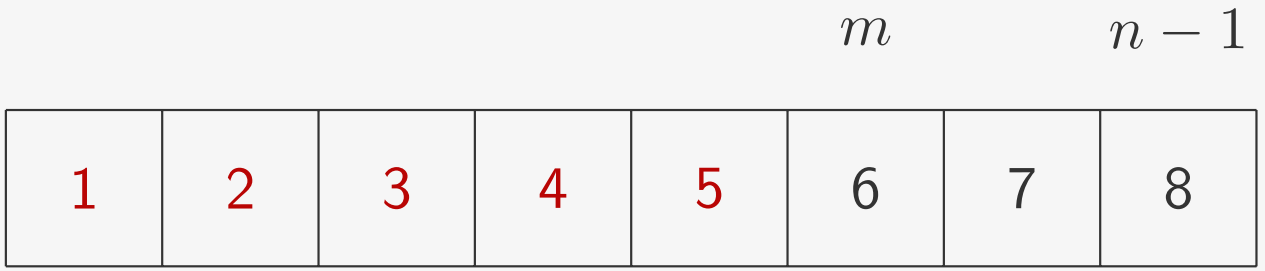


Vamos escreve uma função `permutacoes` que:

- Recebe um vetor com n números distintos:
  - m primeiras posições devem ser mantidas
  - posições de m até $n-1$ devem ser permutadas

- Imprime todas permutações que esse prefixo:
  - 1, 2, 3, 4, 5, 6, 7, 8

Permutações

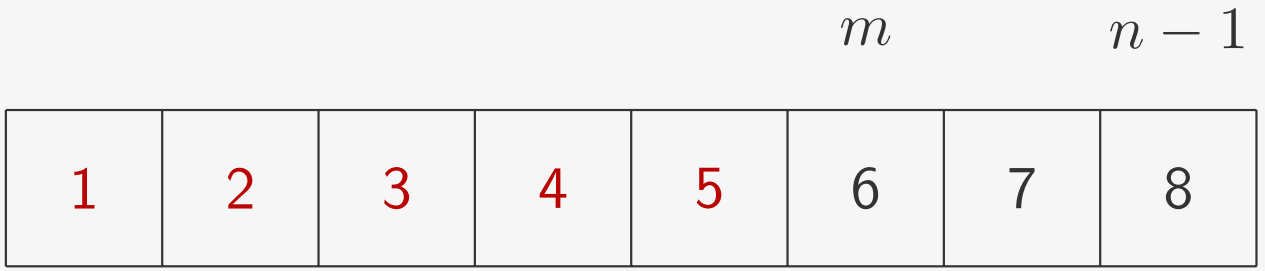


Vamos escreve uma função `permutacoes` que:

- Recebe um vetor com n números distintos:
  - m primeiras posições devem ser mantidas
  - posições de m até $n-1$ devem ser permutadas

- Imprime todas permutações que esse prefixo:
  - 1, 2, 3, 4, 5, 6, 7, 8
  - 1, 2, 3, 4, 5, 6, 8, 7

Permutações

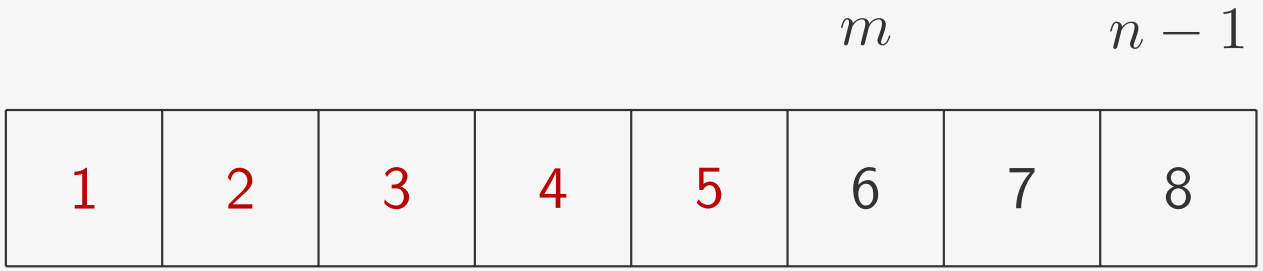


Vamos escreve uma função `permutacoes` que:

- Recebe um vetor com n números distintos:
  - m primeiras posições devem ser mantidas
  - posições de m até $n-1$ devem ser permutadas

- Imprime todas permutações que esse prefixo:
  - 1, 2, 3, 4, 5, 6, 7, 8
  - 1, 2, 3, 4, 5, 6, 8, 7
  - 1, 2, 3, 4, 5, 7, 6, 8

Permutações

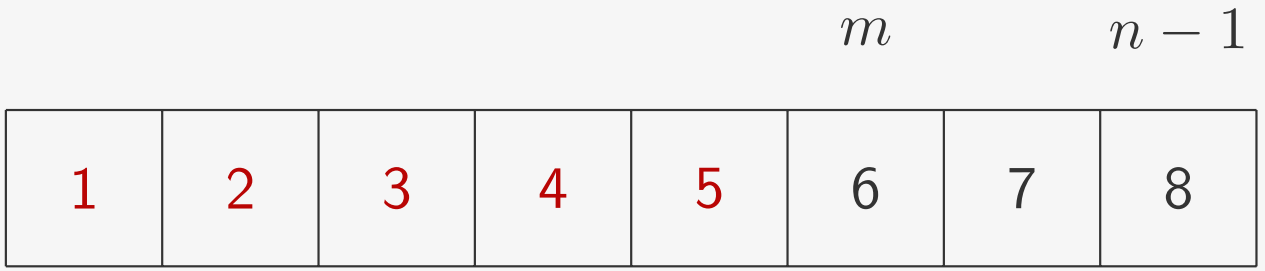


Vamos escreve uma função `permutacoes` que:

- Recebe um vetor com n números distintos:
  - m primeiras posições devem ser mantidas
  - posições de m até $n-1$ devem ser permutadas

- Imprime todas permutações que esse prefixo:
  - 1, 2, 3, 4, 5, 6, 7, 8
  - 1, 2, 3, 4, 5, 6, 8, 7
  - 1, 2, 3, 4, 5, 7, 6, 8
  - 1, 2, 3, 4, 5, 7, 8, 6

Permutações

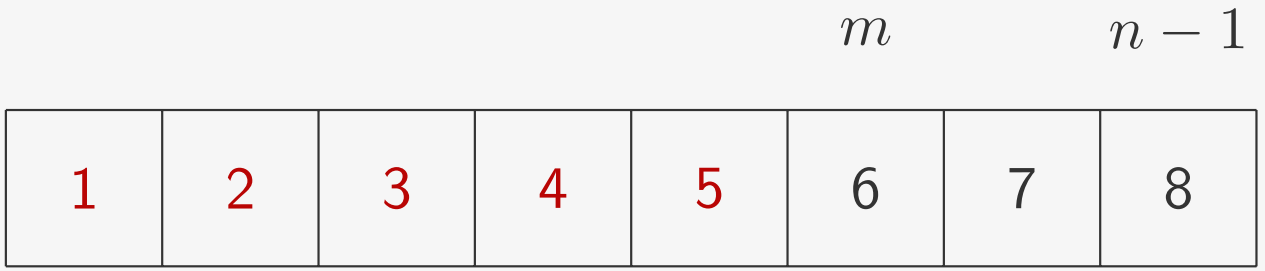


Vamos escreve uma função `permutacoes` que:

- Recebe um vetor com n números distintos:
  - m primeiras posições devem ser mantidas
  - posições de m até $n-1$ devem ser permutadas

- Imprime todas permutações que esse prefixo:
  - 1, 2, 3, 4, 5, 6, 7, 8
  - 1, 2, 3, 4, 5, 6, 8, 7
  - 1, 2, 3, 4, 5, 7, 6, 8
  - 1, 2, 3, 4, 5, 7, 8, 6
  - 1, 2, 3, 4, 5, 8, 6, 7

Permutações

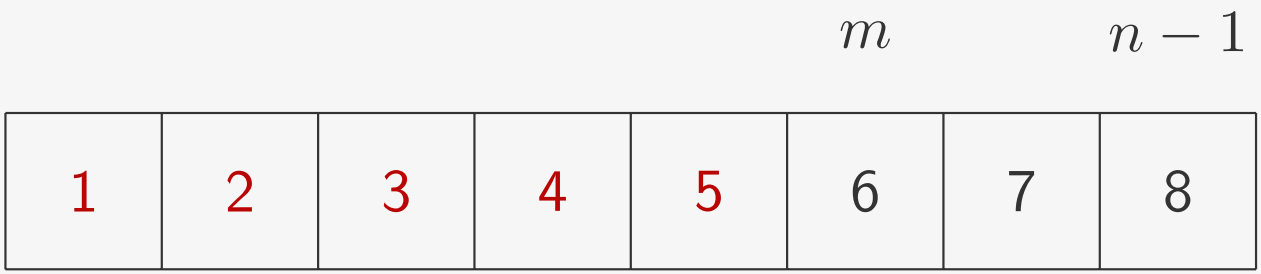


Vamos escreve uma função `permutacoes` que:

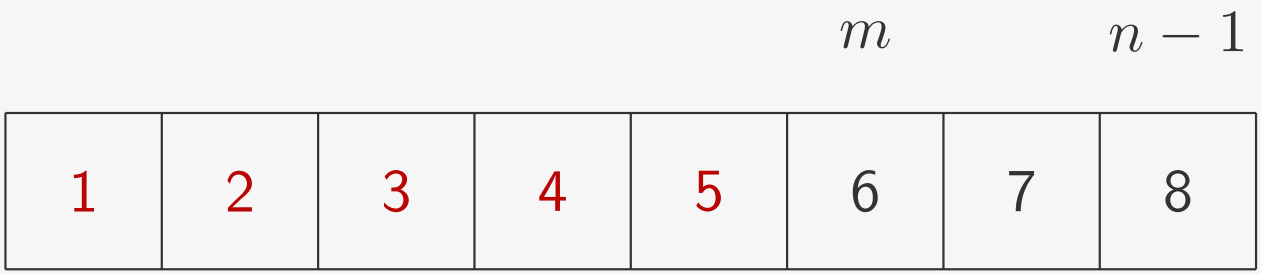
- Recebe um vetor com n números distintos:
  - m primeiras posições devem ser mantidas
  - posições de m até $n-1$ devem ser permutadas

- Imprime todas permutações que esse prefixo:
  - 1, 2, 3, 4, 5, 6, 7, 8
  - 1, 2, 3, 4, 5, 6, 8, 7
  - 1, 2, 3, 4, 5, 7, 6, 8
  - 1, 2, 3, 4, 5, 7, 8, 6
  - 1, 2, 3, 4, 5, 8, 6, 7
  - 1, 2, 3, 4, 5, 8, 7, 6

Gerando permutações

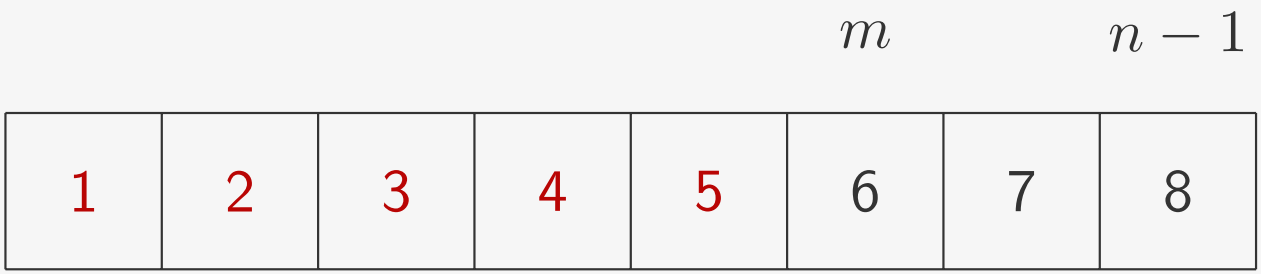


Gerando permutações



Para cada índice i ainda não fixado:

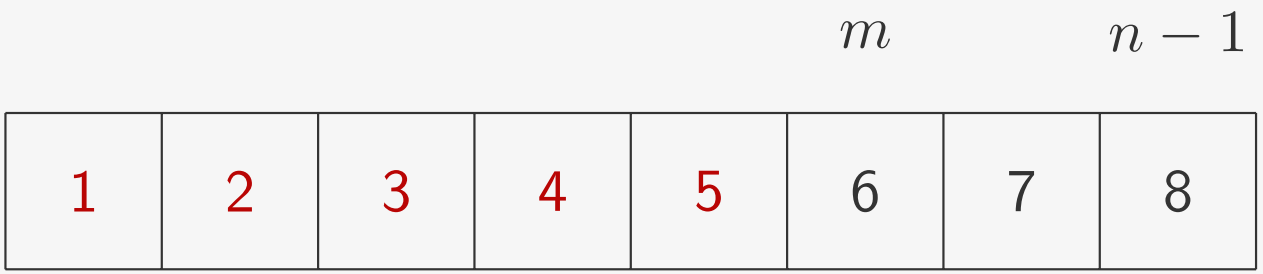
Gerando permutações



Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m

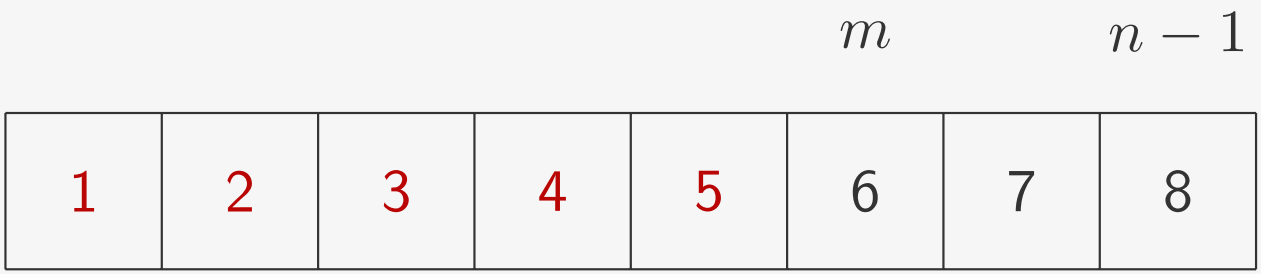
Gerando permutações



Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m

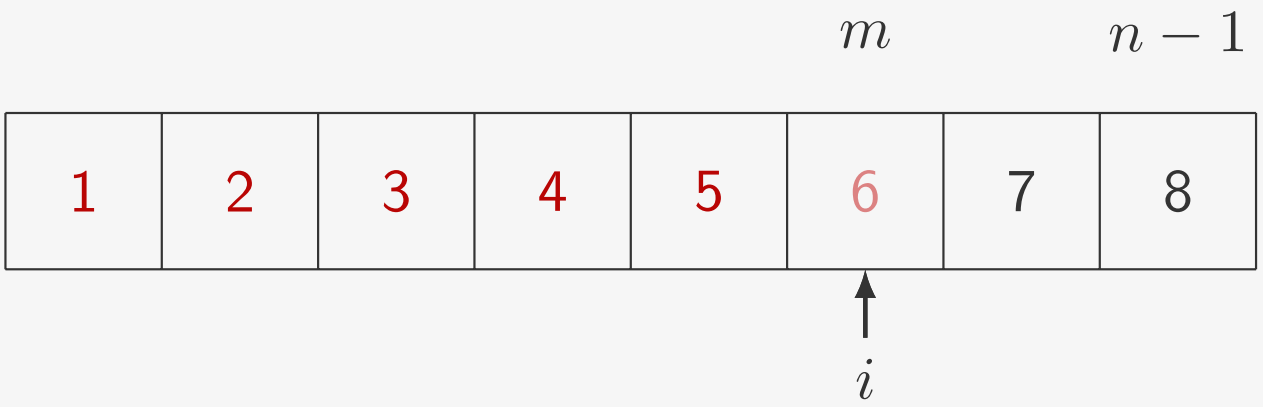
Gerando permutações



Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

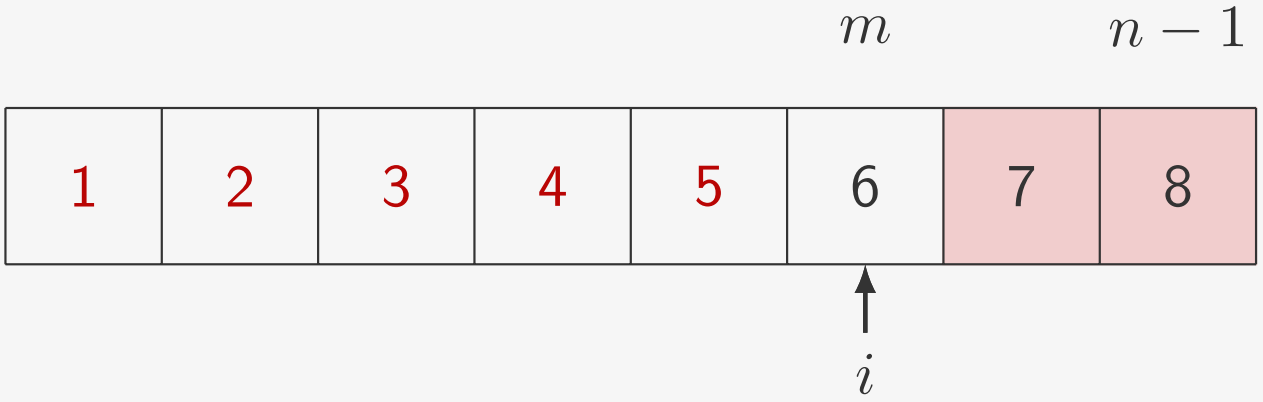
Gerando permutações



Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

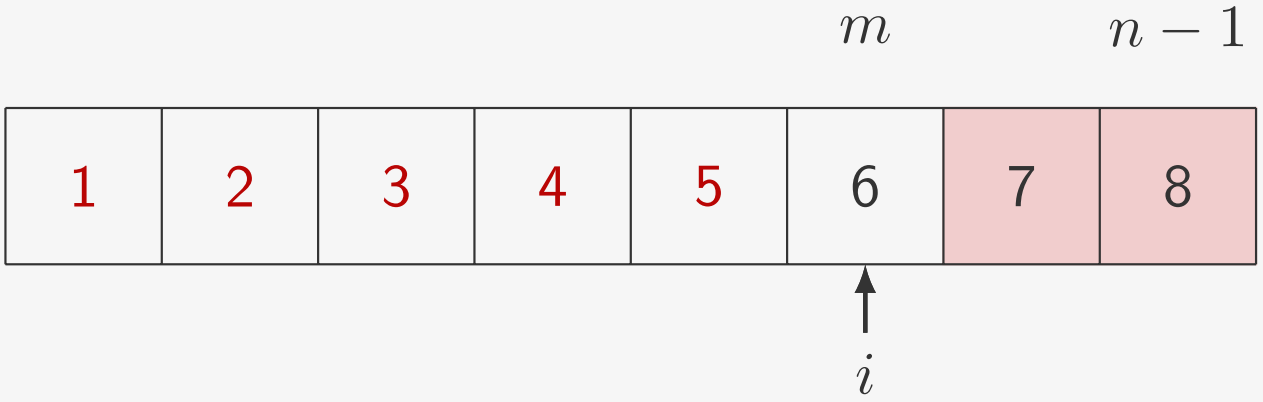
Gerando permutações



Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

Gerando permutações

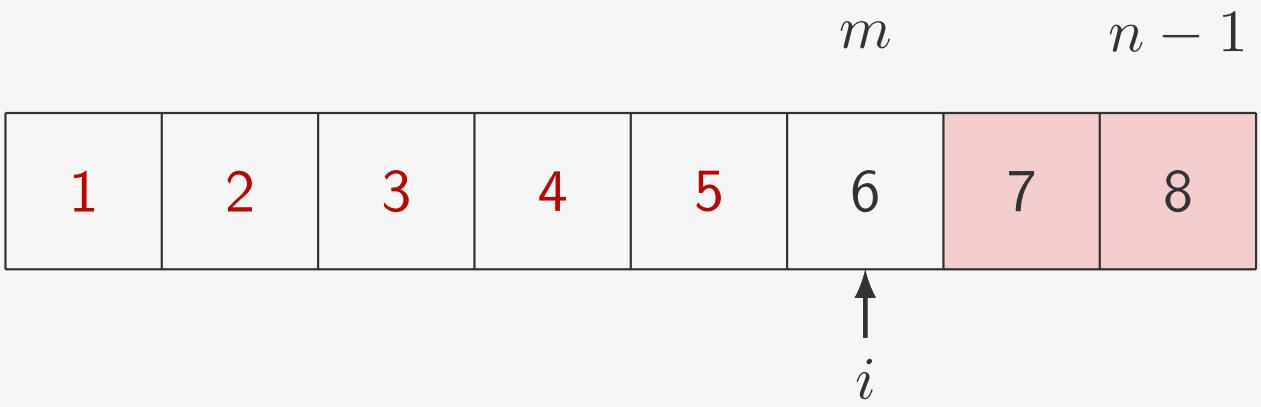


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8

Gerando permutações

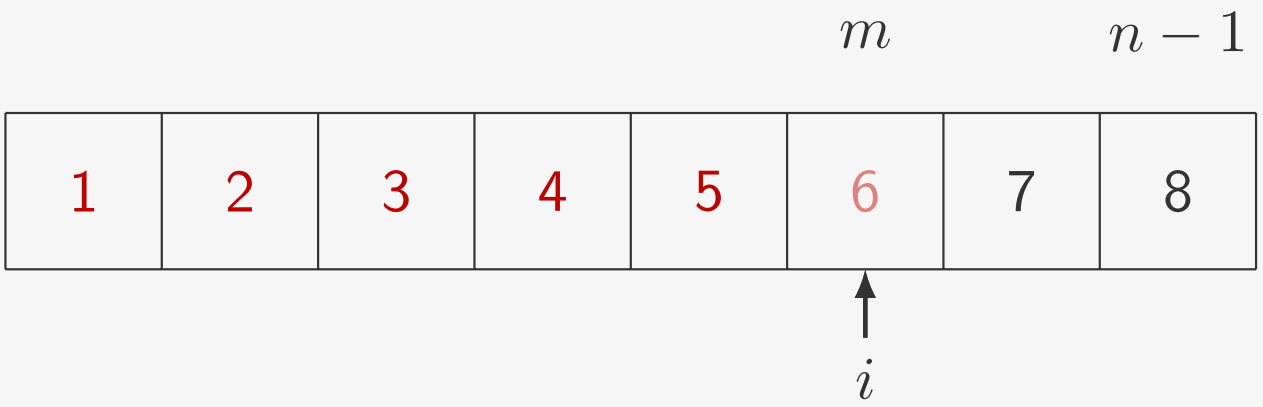


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7

Gerando permutações

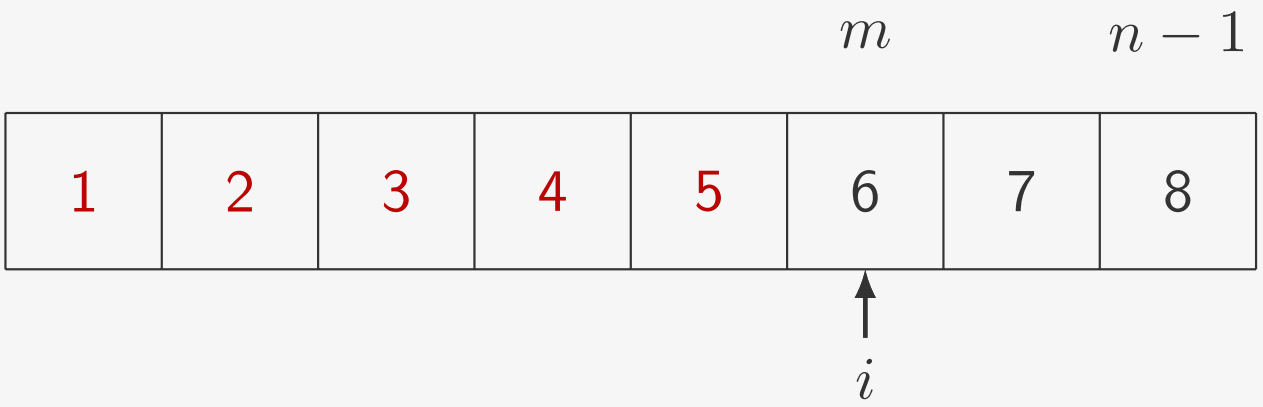


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7

Gerando permutações

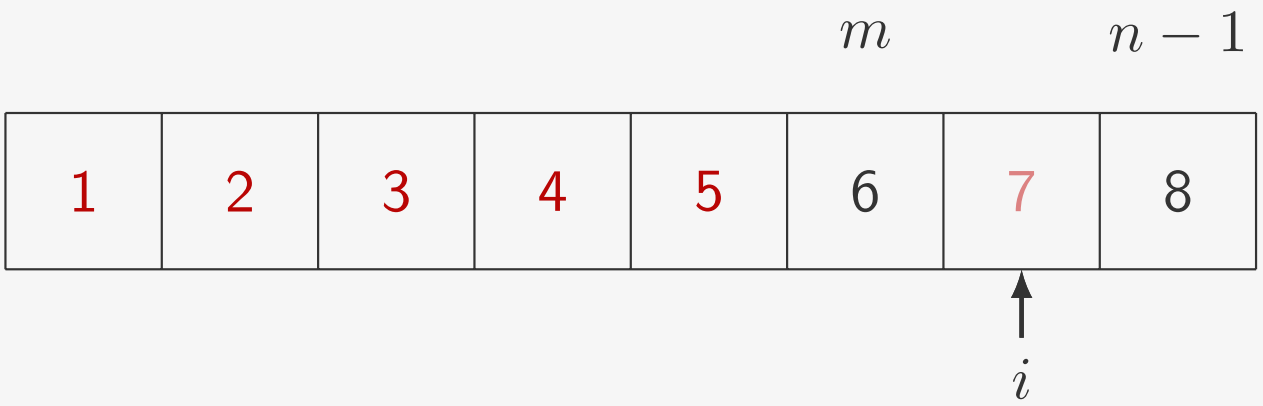


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7

Gerando permutações

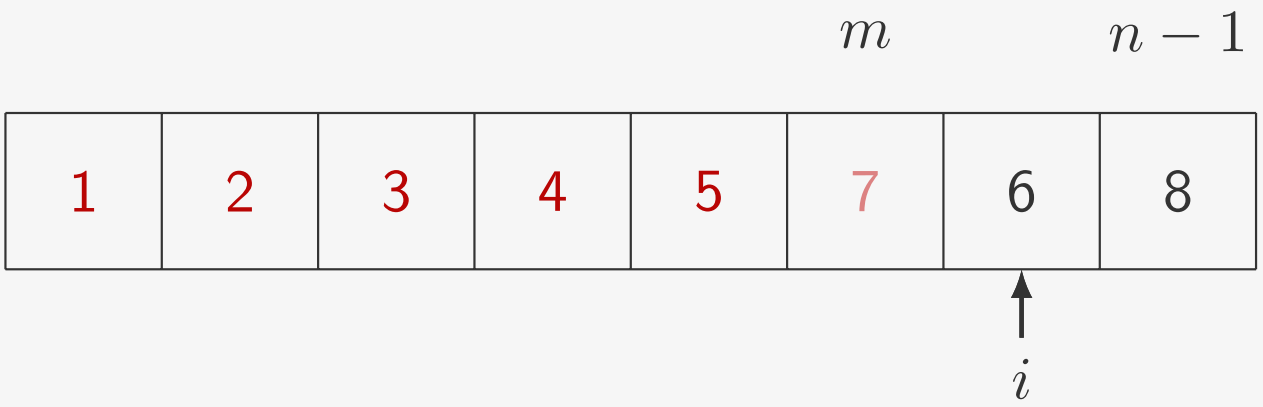


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7

Gerando permutações

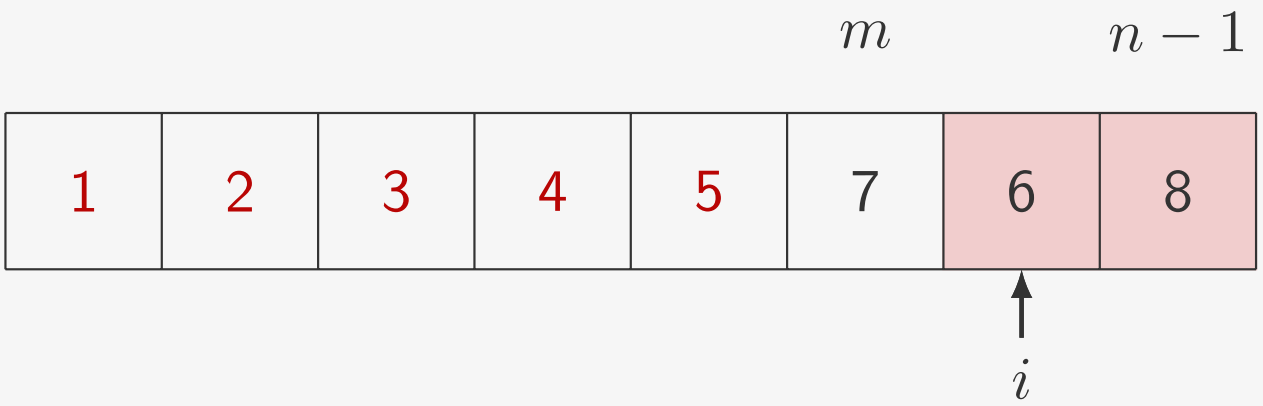


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7

Gerando permutações

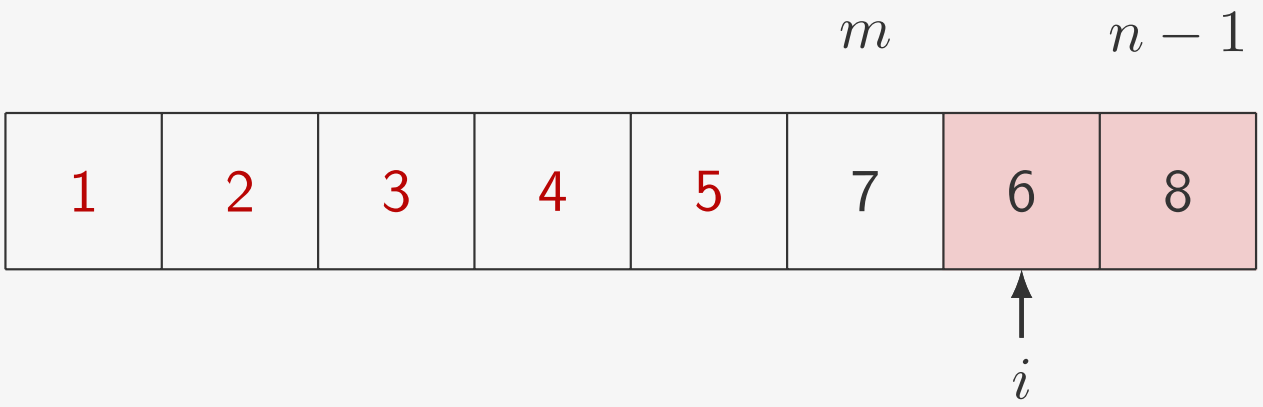


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7

Gerando permutações

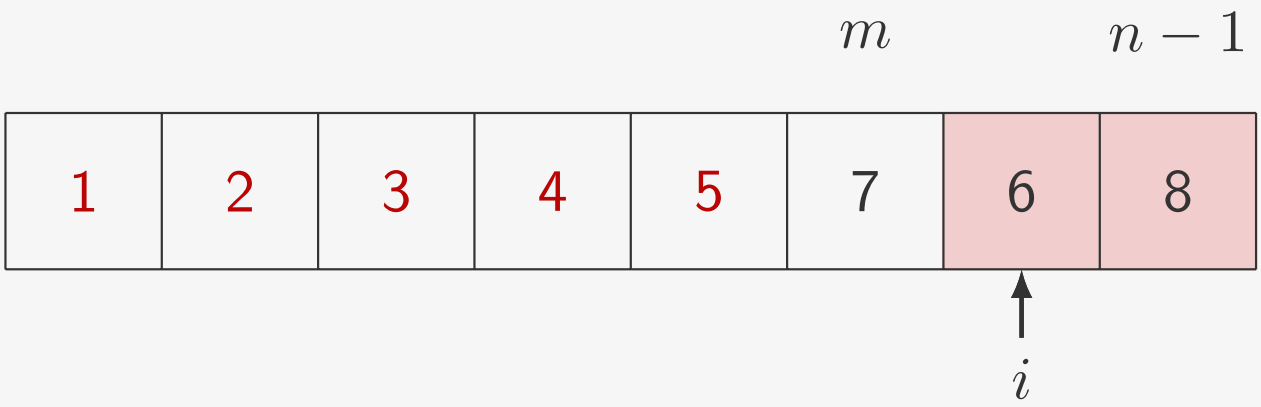


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8

Gerando permutações

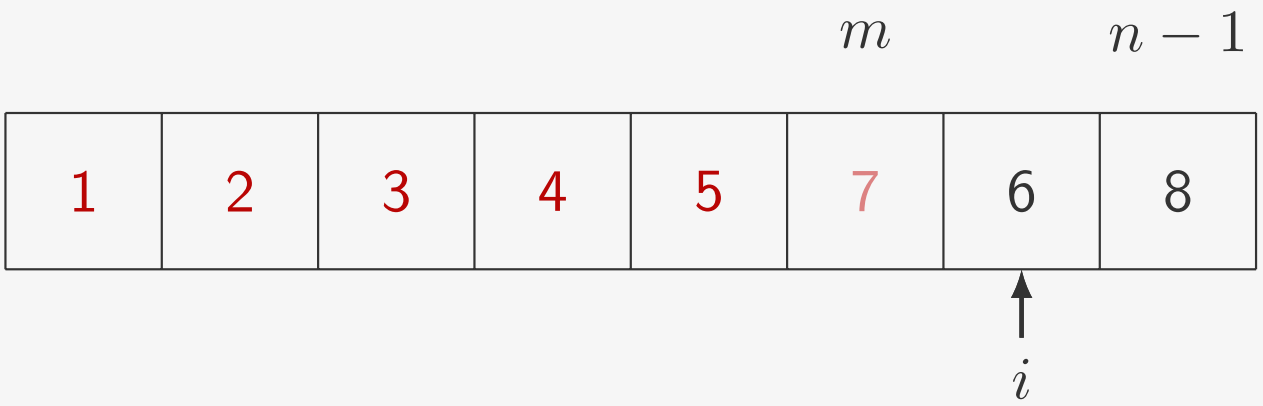


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6

Gerando permutações

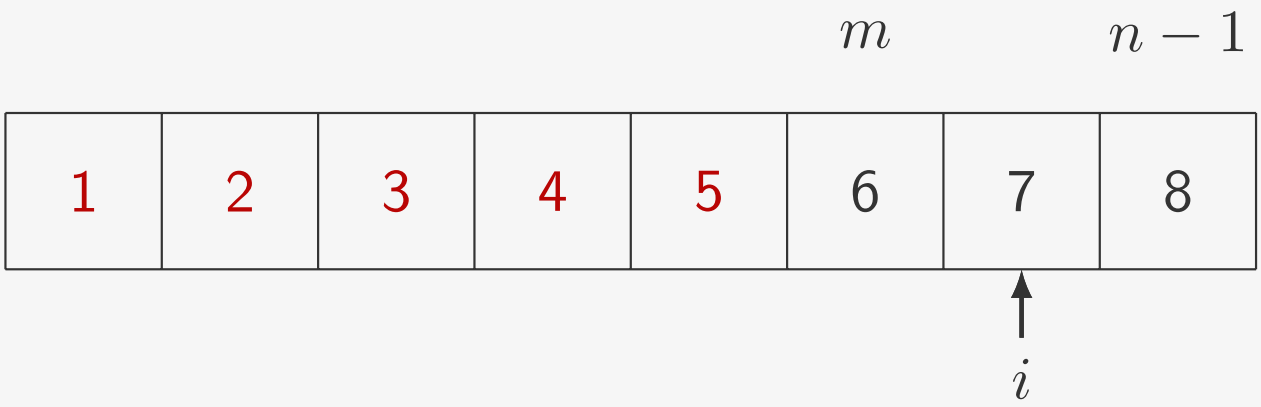


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6

Gerando permutações

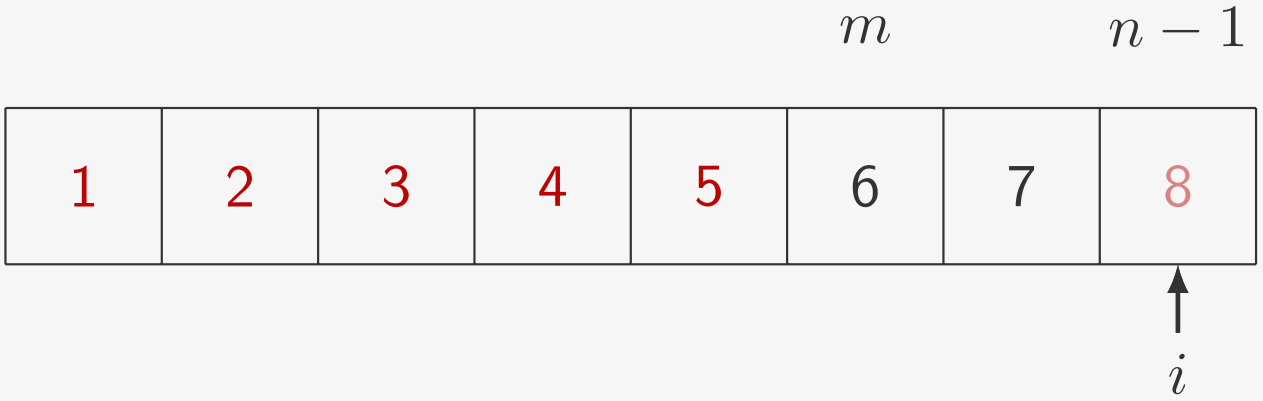


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6

Gerando permutações

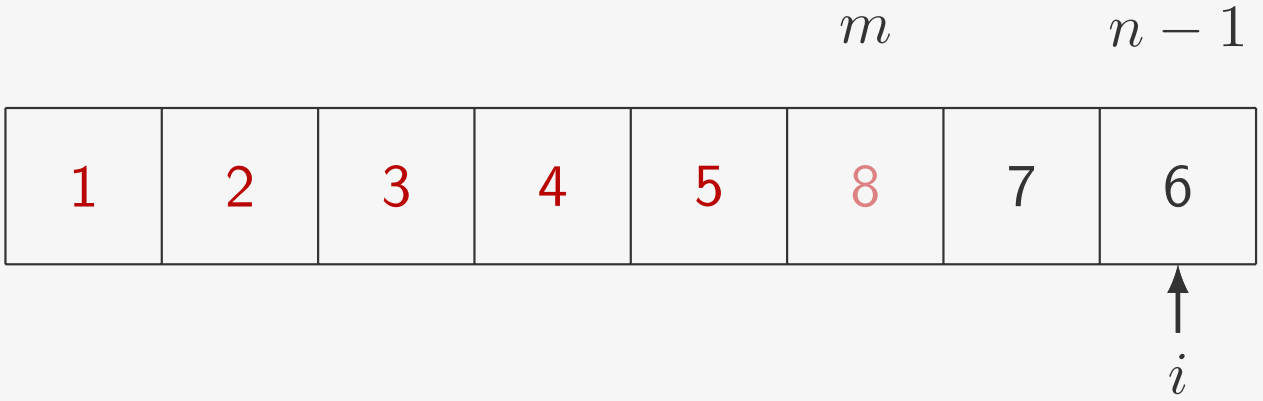


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6

Gerando permutações

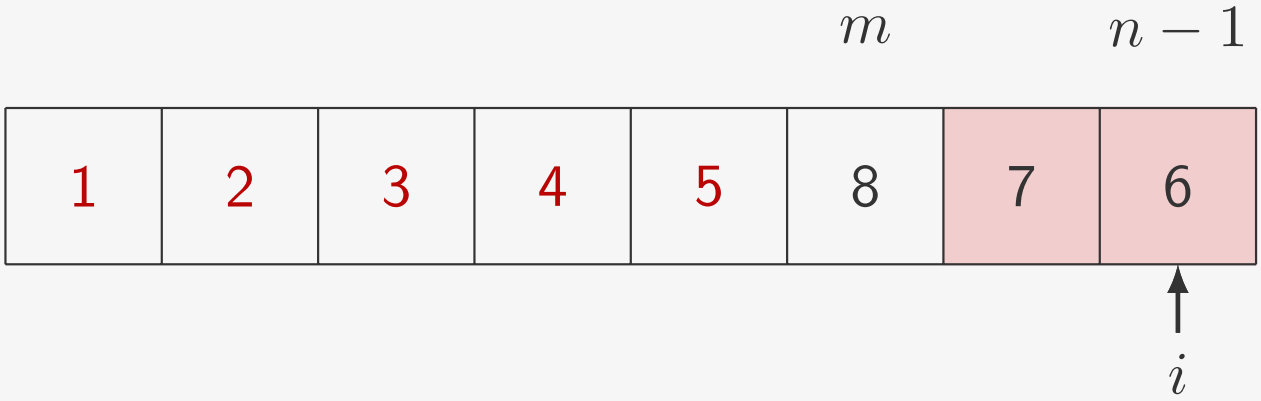


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6

Gerando permutações

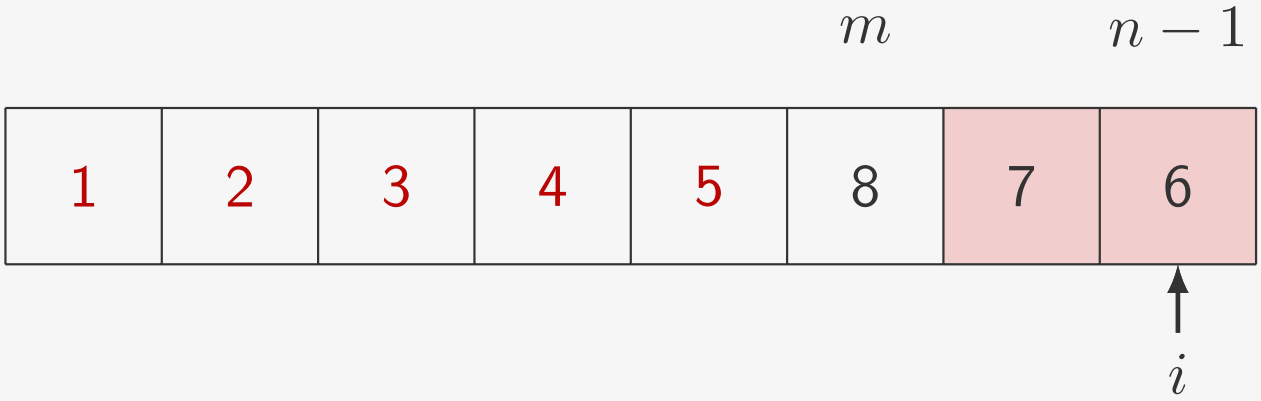


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6

Gerando permutações

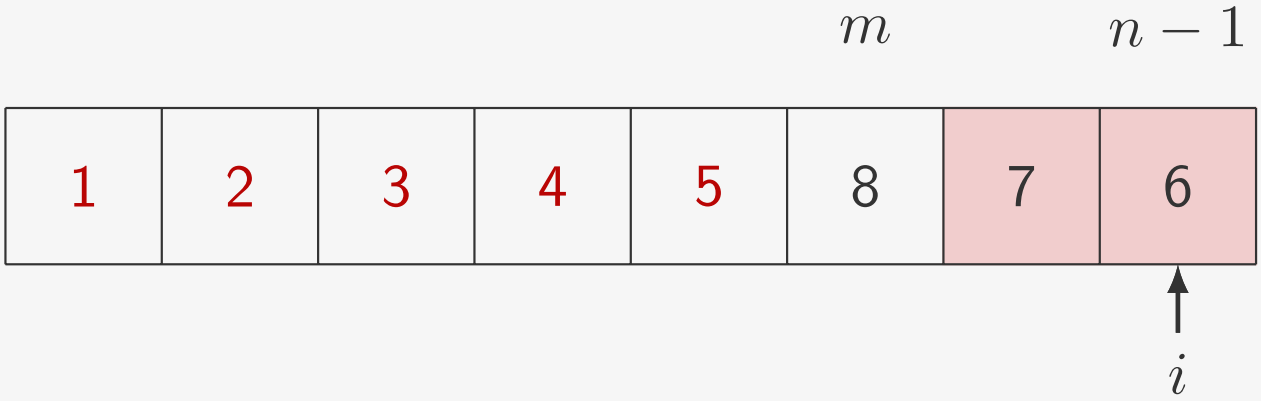


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6
- 1, 2, 3, 4, 5, 8, 6, 7

Gerando permutações

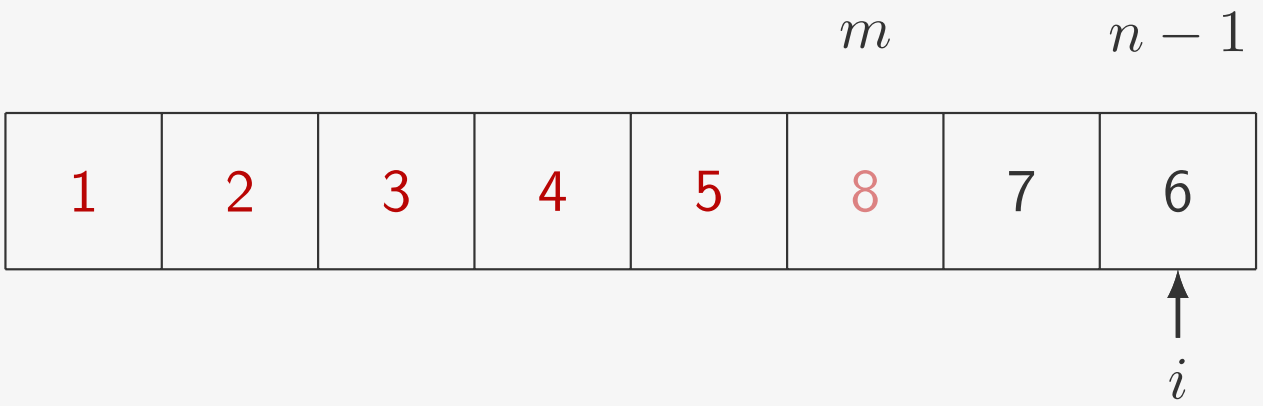


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6
- 1, 2, 3, 4, 5, 8, 6, 7
- 1, 2, 3, 4, 5, 8, 7, 6

Gerando permutações

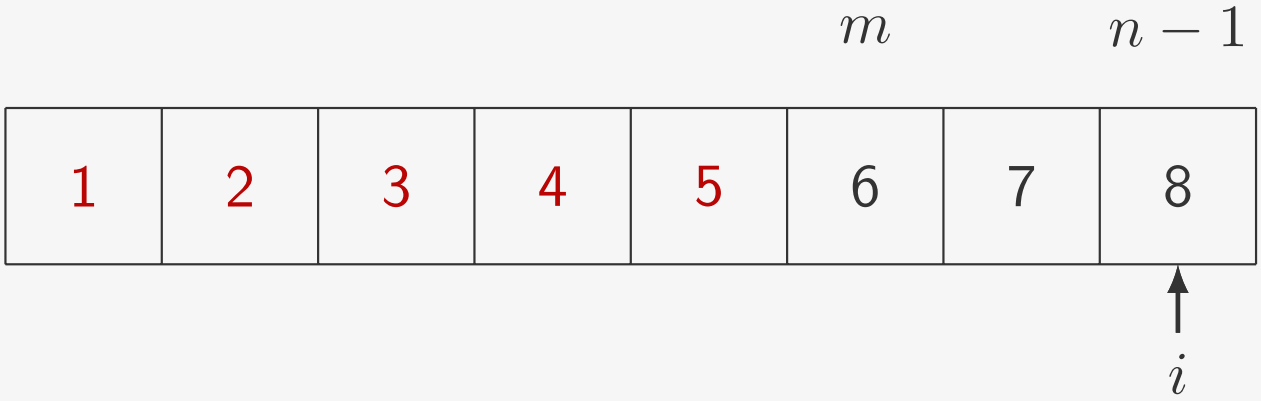


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6
- 1, 2, 3, 4, 5, 8, 6, 7
- 1, 2, 3, 4, 5, 8, 7, 6

Gerando permutações

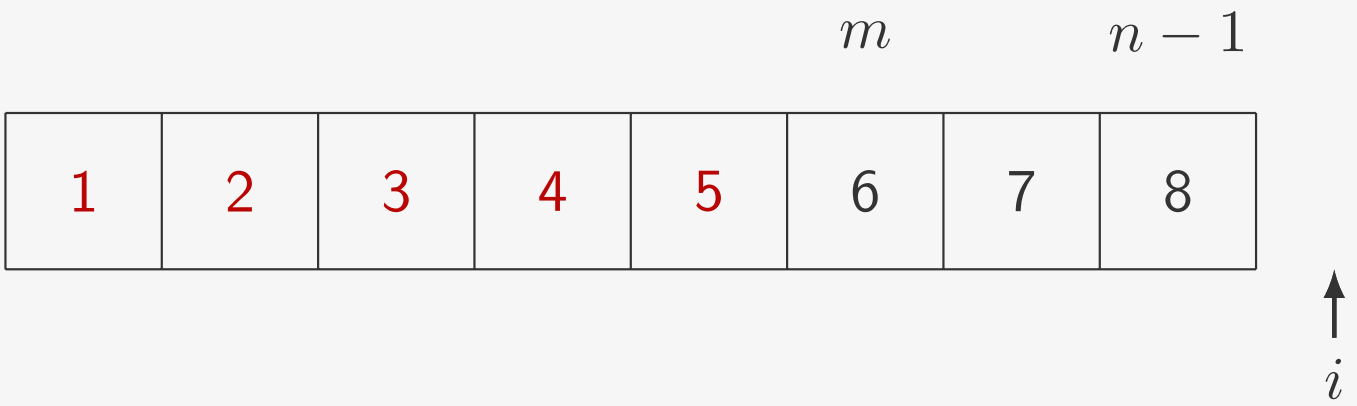


Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6
- 1, 2, 3, 4, 5, 8, 6, 7
- 1, 2, 3, 4, 5, 8, 7, 6

Gerando permutações



Para cada índice i ainda não fixado:

1. trocamos a posição i com a posição m
2. listamos as permutações recursivamente fixando m
3. voltamos às posições originais

- 1, 2, 3, 4, 5, 6, 7, 8
- 1, 2, 3, 4, 5, 6, 8, 7
- 1, 2, 3, 4, 5, 7, 6, 8
- 1, 2, 3, 4, 5, 7, 8, 6
- 1, 2, 3, 4, 5, 8, 6, 7
- 1, 2, 3, 4, 5, 8, 7, 6

Programando

Permutando vetores de tamanho n:

Programando

Permutando vetores de tamanho n:

```
1 void permutacoes(int vetor[], int m, int n) {
```

Programando

Permutando vetores de tamanho n:

```
1 void permutacoes(int vetor[], int m, int n) {
2   // se todo vetor estiver fixo, só há uma permutação
3   if (n == m) {
4     imprimir_vetor(vetor, n);
5     return ;
6   }
7
```

Programando

Permutando vetores de tamanho n:

```
1  void permutacoes(int vetor[], int m, int n) {
2    // se todo vetor estiver fixo, só há uma permutação
3    if (n == m) {
4      imprimir_vetor(vetor, n);
5      return ;
6    }
7
8    // senão, então fixa posição m com cada valor livre
9    for (int i = m; i < n; i++) {
10     troca(&vetor[m], &vetor[i]);
11     permutar(vetor, m + 1, n);
12     troca(&vetor[m], &vetor[i]);
13   }
14 }
```

Programando

Permutando vetores de tamanho n:

```
1  void permutacoes(int vetor[], int m, int n) {
2    // se todo vetor estiver fixo, só há uma permutação
3    if (n == m) {
4      imprimir_vetor(vetor, n);
5      return ;
6    }
7
8    // senão, então fixa posição m com cada valor livre
9    for (int i = m; i < n; i++) {
10     troca(&vetor[m], &vetor[i]);
11     permutar(vetor, m + 1, n);
12     troca(&vetor[m], &vetor[i]);
13   }
14 }
```

Exercício: resolver o problema das damas usando permutação

Backtracking

Backtracking ou retrocesso é um algoritmo genérico, com as seguintes propriedades

Backtracking

Backtracking ou retrocesso é um algoritmo genérico, com as seguintes propriedades

- as soluções são construídas incrementalmente

Backtracking

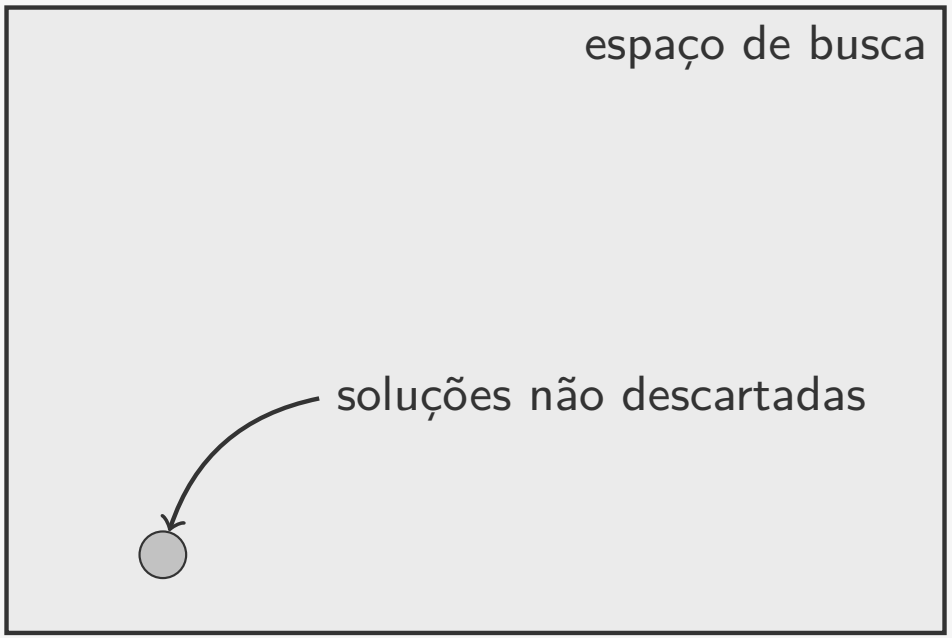
Backtracking ou retrocesso é um algoritmo genérico, com as seguintes propriedades

- as soluções são construídas incrementalmente
- uma solução parcial é descartada tão logo ela se mostre inviável

Backtracking

Backtracking ou retrocesso é um algoritmo genérico, com as seguintes propriedades

- as soluções são construídas incrementalmente
- uma solução parcial é descartada tão logo ela se mostre inviável



Eficiência do Backtracking

- Mais rápido que força Bruta pois eliminamos vários candidatos de uma só vez

Eficiência do Backtracking

- Mais rápido que força Bruta pois eliminamos vários candidatos de uma só vez
- Implementação simples, mas pode ser lento para problemas com muitas soluções parciais

Eficiência do Backtracking

- Mais rápido que força Bruta pois eliminamos vários candidatos de uma só vez
- Implementação simples, mas pode ser lento para problemas com muitas soluções parciais

Como fazer um algoritmo de Backtracking rápido?

Eficiência do Backtracking

- Mais rápido que força Bruta pois eliminamos vários candidatos de uma só vez
- Implementação simples, mas pode ser lento para problemas com muitas soluções parciais

Como fazer um algoritmo de Backtracking rápido?

- O algoritmo para verificar solução parcial deve ser:

Eficiência do Backtracking

- Mais rápido que força Bruta pois eliminamos vários candidatos de uma só vez
- Implementação simples, mas pode ser lento para problemas com muitas soluções parciais

Como fazer um algoritmo de Backtracking rápido?

- O algoritmo para verificar solução parcial deve ser:
  - Bom: evita explorar muitas soluções parciais

Eficiência do Backtracking

- Mais rápido que força Bruta pois eliminamos vários candidatos de uma só vez
- Implementação simples, mas pode ser lento para problemas com muitas soluções parciais

Como fazer um algoritmo de Backtracking rápido?

- O algoritmo para verificar solução parcial deve ser:
  - Bom: evita explorar muitas soluções parciais
  - Rápido: processa cada solução rapidamente

Aplicações para Backtracking

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo

- Problemas de Otimização Combinatória

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo

- Problemas de Otimização Combinatória
  - Conseguimos enumerar as soluções do problema

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo

- Problemas de Otimização Combinatória
  - Conseguimos enumerar as soluções do problema
  - Queremos encontrar a de valor mínimo

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo
- Problemas de Otimização Combinatória
  - Conseguimos enumerar as soluções do problema
  - Queremos encontrar a de valor mínimo
- Programação Lógica (Prolog, por exemplo)

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo
- Problemas de Otimização Combinatória
  - Conseguimos enumerar as soluções do problema
  - Queremos encontrar a de valor mínimo
- Programação Lógica (Prolog, por exemplo)
  - Prova automática de teoremas

Exercício

Crie um algoritmo que, dado n e C, imprime todas as sequências de números inteiros positivos $x_1, x_2, \ldots, x_n$ tal que

$$x_1 + x_2 + \cdots + x_n = C$$

a) Modifique o seu algoritmo para considerar apenas sequências sem repetições

b) Modifique o seu algoritmo para imprimir apenas sequências com $x_1 \leq x_2 \leq \cdots \leq x_n$